Assignment 2 – Games and Constraint Satisfaction Programming (CSP)

Preliminaries

The programming language for this assignment is Python 3 and Prolog. The assignment should be submitted through ilearn2 no later than the 29th of September at 23.59 (CET).

Games

In the file “PFAI_Assignment_2a.zip” found in the ilearn2 course page, there are three files: ‘run_assignment_2.py’, ‘four_in_a_row.py’, and ‘game_node_and_game_search.py’. The game you shall implement is “Four in a row”, see figure 1. The games are a two-person zero-sum game with full information. The aim of the game is to get four chips in a row, either: horizontally, vertically or diagonally. The board consists of 6 row and 7 columns. The file ‘four_in_a_row.py’ contains a skeleton code for defining the game, the board of the game is represented as a list containing lists of all columns. You’ll need to define all the legal **actions** of the game and two (2) checks to see if the game is in a **terminal state** (won, draw or loss). You also need to define a **print function** of the board, named pretty_print(). Below is an example of how the print out should look like.

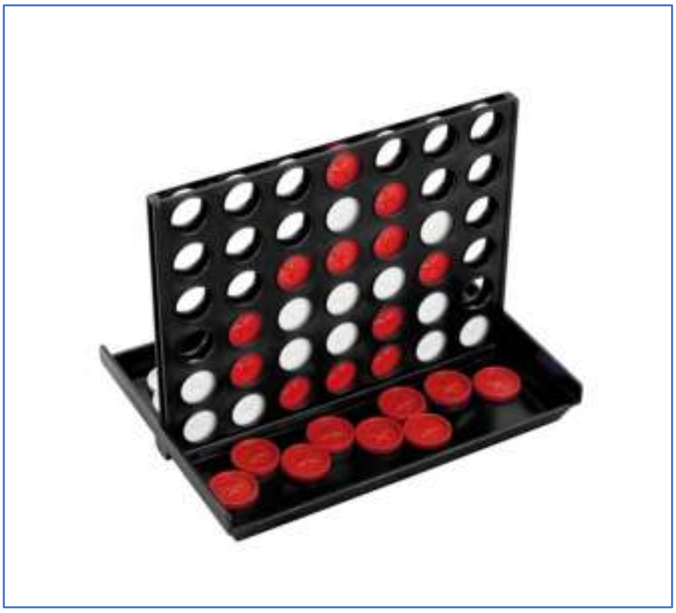

Figure 1:Four in a row

```
_ _ _ w _ _ _
_ _ r r _ _ _
_ _ w r _ _ _
_ _ r r w _ _
_ _ r w w w _
_ _ r r w w _
```

The file ‘run_assignment_2.py’, contains a game loop is that is lacking the function that **receives input** (get_player_input) from the player of the game, this needs to be addressed by you. Note that you here need to cope with sloppy players and should only accept input that are **legal** according to the rules of the game. You are now ready to play your first game against the computer, give it a try!

TODO: CODE

- **Complete** the code for the actions, i.e., a list of legal actions
- **Complete** the code for the terminal state.
- **Complete** the code for pretty_print
- **Complete** the code for get_player_input

In 'game_node_and_game_search.py' you have a complete minimax algorithm, and also a definition of a game node. As you might notice when you play the game it does not play very cleverly, even when you increase the depth. Hence, you'll need to **modify the algorithm such that implements alfa-beta pruning**. After this, play again, does it play much better? Write down **your answer and your thoughts** regarding this issue, in a text document.

To make the AI play better you will now **implement an evaluation function** (eval) which should be used when the *depth limit is reached* (instead of the zero (0) value if non terminal nodes are found). You can design this function freely, but a word of caution, try not to be overly specific. As a rule of thumb when playing four in a row, positions in the middle of the board are considered more valuable than those at the edges. A tip is to count chips value more towards the center of the board. Try to play again! **Any difference**? Write down your answer in the text document. Finally, add a **time-based stopping criterion** instead of relying on the depth criterion. So that if a time has been set, it should supersede the depth criterion (i.e., do not remove the possibility of using a depth criterion). As in the previous assignment the `process_time` library is recommended.

TODO: CODE and EXPERIMENT

- **Modify** the minimax code such that implements alfa-beta pruning.
- **Write down** the result of your subjective comparison between minimax and minimax with alfa-beta pruning. Note, if you have time, you can actually play the two versions against each other, for a better comparison.
- **Add** an evaluation function, eval.
- **Write down** your thoughts after playing against the AI with the eval function.
- **Add** a time based stopping criterion.

Now it is time to look at *Monte Carlo Tree Search*, in 'game_node_and_game_search.py' you find the function mcts. You need to define what information you need to keep track of in the GameNode for each state of the game Then you will need to define the functions of the main loop in the mcts function. Recommended *playout policy* is random selection, hence use the `random` library. The recommended *selection policy* is as follows: if a node has actions left, randomly choose one of them, if all actions are done, select the child with the highest UCB1 scores of all children.

Implement the **select** function who selects a node to expand and the **expand** function that creates new children nodes. **Simulate** that given a node simulate a playout until a terminal state and return the result and **back_propagate** which keeps tracks of the number of wins and losses for each node. Finally, there is the function **actions** which returns the best move given the information the AI gathered (simulated) so far. Note that the node (GameNode) needs to be updated with information to keep track of: **parent, playouts, wins, successors (list), move** and **actions_left.** Try the MCTS game out, **is it playing better than minimax with alfa-beta pruning**?

TODO: CODE and EXPERIMENT

- **Update** the node definition to keep track of: parent, playouts, wins, successors (list), move and actions_left.
- **Implement `select`** which preferably is written recursively, stopping condition is if the current node has actions left, if so, stop recursion and return this node. In the recursive case you need to utilize ucb1 selection of nodes.
- **Implement `expand`** - remember to check if terminal node is reached. The method should return a child to current leaf.
- **Implement `simulate`** - remember to check if terminal node is reached, else expand using random selection until a terminal state is found and return win or loss
- **Implement `back_propagate`** - keep track of which player turn it is to move, so you can update the node with the correct counts of wins.
- **Implement `actions`** function which return a move, i.e. the best move according to the playouts that have been simulated so far.
- **Write down** your thoughts after playing against the MCTS AI.

Constraint Satisfaction Problems

Using SICStus prolog and the CLPFD library you shall now look in to the following classical problem called the zebra puzzle. At your disposal you have a code skeleton, zebra_puzzle.pl (in PFAI_Assignment 2b.zip) to get you started. Given the following information:

There are five houses.

The English man lives in the red house.

The Swede has a dog.

The Dane drinks tea.

The green house is immediately to the left of the white house.

They drink coffee in the green house.

The man who smokes Pall Mall has birds.

In the yellow house they smoke Dunhill.

In the middle house they drink milk.

The Norwegian lives in the first house.

The man who smokes Blend lives in the house next to the house with cats.

In a house next to the house where they have a horse, they smoke Dunhill.

The man who smokes Blue Master drinks beer.

The German smokes Prince.

The Norwegian lives next to the blue house.

They drink water in a house next to the house where they smoke Blend.

Modify the code skeleton so that the program will give you the answer to who owns the zebra and more! Hint: Divide all information into: house colors, pets, smokes etc. Also note that, ***#\/*** defines ***or*** for the clpfd solver and ***#= assignment.*** Please note, the ***full answer*** of who is living where, what they smoke, drink and their pets in your *text document.* The hand-in should also contain the complete code, that should be able to run with the **|?- zebra.** command.

Summary of assignment

1. Your groups assignment should contain the code that you have produced:
2. ‘run_assignment_2.py’ – containing additional initialization and call for mcts.
3. ‘four_in_a_row.py’ - completed according to assignment.
4. ‘game_node_and_game_search.py’ - completed according to assignment.
5. text document – Notes from your experiments, in the same order as they are requested in this document.
6. Zebra_puzzle.pl - completed according to assignment.
7. Add all files above into a zip file with the last names of both group members, like ‘LastName1_LastName2_ass2.zip’ and submit it in ilearn2. Note that only one submission should be sent in from each group, which member who send it in do not matter.