COGNI CHESS DOCUMENTATION

By Ioannis Katelouzos.

Rev : 1.00

Contents

1.Introduction

This is the documentation for CogniChess. CogniChess is a game available in various platforms and can be downloaded from the developer's Web site : www.diversemechanics.com. The reading of this documentation is highly advisable, before attempting to play the game.

First of all this is an attempt to create an application exploiting the complexity of Boolean Networks (Par.2.I). It is understandable that making a game that demands some studying will not make it much popular, but still, apart from a game this wants to be an application that will help people explore and get inspired by BNs. If this game helps at least one person to see deeper implications and effects of these networks, then I will consider this game successful.

A secondary objective is to apply reinforcement learning techniques in order to implement an AI capable of learning this new kind of game. There is the last section that is dedicated to this part of the project.

Please feel free to contact me for any questions or comments.

2.The Network

This documentation is not meant to make a thorough scientific presentation of the subjects, but rather a simple introduction of the notions that are essential, for the understanding of the rules of the game. The reader that is interested in further details will have no trouble finding them on the internet, as these are widely known and documented concepts.

I.Boolean Networks

Boolean networks are directed networks where each node represents a Boolean function. Incoming edges are the inputs to the function which output is applied to the outgoing Edges, continuing the evolution of the network. Obviously Inputs and outputs can only be Boolean variables, which we can consider to represent the excited or refractory state of the edges and nodes.

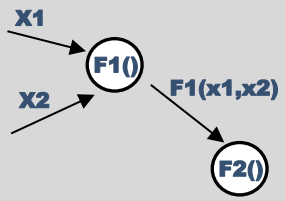


FIGURE 2-1: GENERIC BOOLEAN NETWORK

A key propriety of the network is its number of states. If a network has N nodes, then the whole network can move in a state space of size 2^N.

In some bibliography you might come across the notion of random Boolean networks. These networks are normal BNs where the functions and starting conditions are chosen randomly from all the possible functions and space state respectively.

- Example

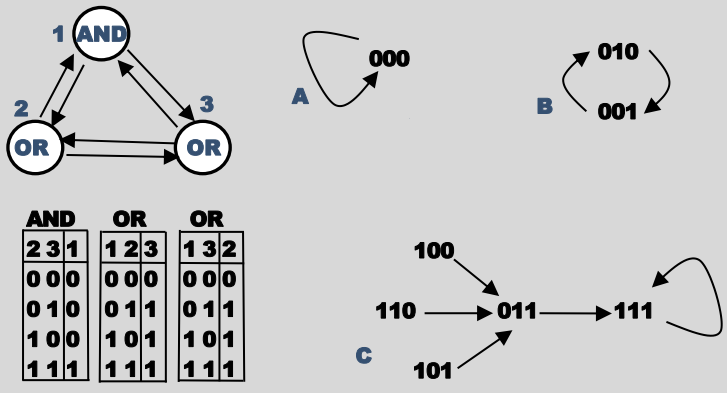


FIGURE 2-2 : A BOOLEAN NETWORK WITH N=3. ON THE LEFT IS THE DEFINITION OF THE NETWORK. ON THE RIGHT THE COMPLETE STATE SPACE WITH THE RELATED TRANSITIONS.

Here we will present a quick example in order to have a visual reference. As you can see in Figure 2-2 : A Boolean network with N=3. On the left is the definition of the network. On the right the complete state space with the related transitions.we have a network with N=3 and $\Omega=2^3=8$ states. We Can note how applying a

different starting condition will make the network evolve through different sequences.

II.Cycles and attractors

We can see that in this example there are three detached sequences (A, B and C). For each of these sequences there is a cycle associated. (000 for A, 010-001 for B and 111 for C). We can logically deduce that each of these sequences, whatever the network and whatever the sequence will always have one and only one such cycle associated. Such cycles act as attractors for the initial condition and this is the correct way to use the terms. In CogniChess we will somehow misuse the term Attractor and will use it to refer both to the whole sequence and the set of the edges and nodes that get excited during the whole cycle. We will do the first in order to distinguish the two in the game, and the later because it seems visually logical even though strictly speaking the attractor refers to the state space and not the network topology. In the second case we will refer to it as the 'visual attractor' to make it more clear.

III.Memory and cognition

It is useful at this point to mention very superficially how scientist relate the memory and cognition to networks.

Let's consider the network in Figure 2-3, and say that the functions in the nodes and the initial condition (C_i) are such that we have an attractor in (b,c,d) that we will denote as C_i[b,c,d]. This means that our network given $\Omega=2^N$ can store as much data as are the attractors that the network can support. One would like to have as many attractors as possible to be able to store as much data as possible and this as we will see is one of the goals in CogniChess. It is also worth noting how efficient networks are for storing data, as an elementary block (say edge bd) can be reused for different data, as in C_i[b,c,d] and maybe C_a[b,d,a]. What's more, as we will see in Par. 3.I the same visual attractor (say [b,c,d]) can have different energies resulting from different cycles and thus two different data can use exactly the same set of elementary blocks (edges). That is we can have C_i[b,c,d] and C_k[b,c,d].

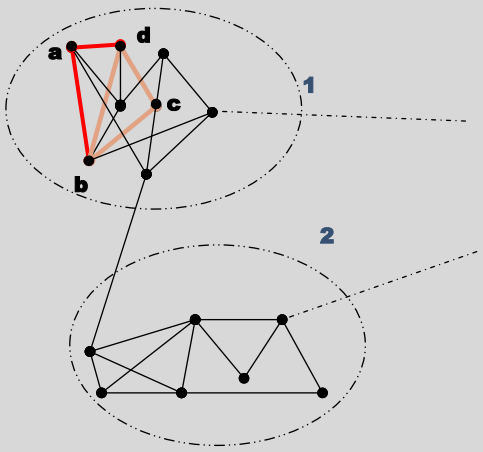


FIGURE 2-3 : MEMORY ELEMENTS IN NETWORKS

Finally, if we want to make an analogy with the animal brain, we can start off by considering that a visual attractor represents a concept and attractors that have common or topologically close edges represent similar concepts. i.e.

C_i[b,c,d] can be ‘cold’, C_k[b,c,d] ‘cool’ and C_a[b,d,a] ‘breeze’. This would make the agglomeration 1 represent ‘low temperatures’ while agglomeration 2 could be ‘high temperatures’. Both of them could be part of an even wider concept as ‘temperatures’. The concept ‘ice’ in this case would activate many attractors across the brain including C_i[b,c,d]. Of course increasing the number of Ω by many orders of magnitude, as it is the case in a brain, can unlock immense possibilities.

3.The Game

I.The CogniChess network

- ### Nodes and edges

In CogniChess the network used is a normal BN, the function used, is the same on every node and it is the following:

$$F_\alpha(\sigma_{1,,}\sigma_i) = \begin{cases} true\ if \sum \sigma_k - \sum \sigma_m > NodeThresh \\ false\ otherwise \end{cases}$$

Where σ_i are all the inputs to the node α, σ_k are all the excited excitatory edges and σ_m are all the excited inhibitory edges.

Nodes have a variable parameter which is $NodeThresh$ and edges have a Boolean value that describe if the edge is excitatory or inhibitory.

- ### Energy

You remember how we described earlier that a visual attractor can appear from different initial conditions, which will result in different cycles and thus energies. Returning to the network in Figure 2-3, when we say that the attractor [b,c,d] will appear, we mean that throughout the cycle, nodes b,c and d are excited at least once. If a node is not excited during the cycle, then it does not take part in the visual attractor. In the same way, in Figure 2-2 that we replaced here for simplicity, we can see for example that cycle A will not excite any node, thus there is no visual attractor at all. Cycle B will excite only nodes 2 and 3 and so the visual attractor will be the edge (2,3) and its nodes. Finally cycle C excites all nodes and as a result the visual attractor will include all elements in the net.

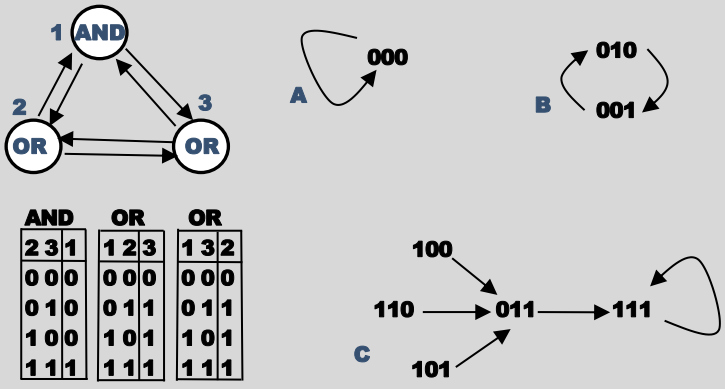


FIGURE 3-1 : A BOOLEAN NETWORK WITH N=3. ON THE LEFT IS THE DEFINITION OF THE NETWORK. ON THE RIGHT THE COMPLETE STATE SPACE WITH THE RELATED TRANSITIONS

Now if we try to associate some energy to the cycles, we can say that an excited node consumes some energy. Cycle A does not excite any nodes so E_A=0. On the opposite side is energy for cycle C that keeps all nodes excited in every step so E_C=1. The case for cycle B is less immediate. We have to count the times a node is excited throughout the cycle and take the average. This gives $E_B = \frac{B_2+B_3}{NumStates} = \frac{\frac{(1+0)}{2}+\frac{(0+1)}{2}}{2} = 0.5$.

Finally, in CogniChess we don't use cycle energies directly, but rather the average energy of the whole net. This is defined as a weighted average, as follows:

$$E_{Av} = \frac{\sum E_i * N_{si}}{N_S}$$

Where E_i is the energy of cycle i, N_{si} are the states in cycle i and N_s is the total number of states in all cycles. For our example we will have : $E_{Av} = \frac{(0*1)+(0.5*2)+(1*1)}{4} = 0.5$.

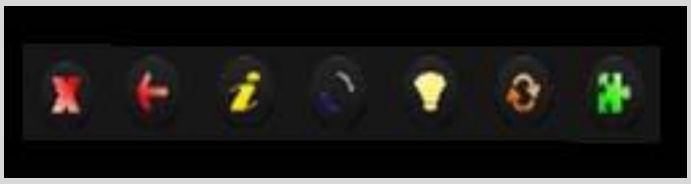
II.Central Panel

This is the main panel of the game and it is where you can view and select the levels to play.

FIGURE 3-2 : THE CENTRAL PANEL.

1. You will be able to select the type of game between tutorial, sandbox, solo and against the AI.
2. Quick description of what you are selecting.
3. Sort the levels by number of nodes, mean energy, number of attractors, type of game or state of the level.
   - There are three types of games for Solo (modify Attractors, biggest Cycle and Energy), two for the AIOP (Constructor, Destroyer) plus the sandbox.
   - Levels may be in four different states. ('never played'-Black, 'tried and failed'-Red, 'Won by watching the solution'-Yellow and 'won'-Green)
4. The list of all available levels.
5. Tweak the volume of effects and music.

III.Action Panel



The action panel can be found on the top-right corner of the screen and consists of seven buttons from the left to the right they are:

FIGURE 3-3 : THE ACTION PANEL.

1) ‘X’ exit the game. Remember to use this button if you want your progress to be saved.
2) ‘Back’ return from the level to the main screen
3) ‘i’ opens this document.
4) ‘Tooltips’ cycles through the tooltips with the IDs of the net objects.
5) ‘Solution’ Shows the solution on the History panel.
6) ‘Reset’ Resets current level.
7) ‘Play’ Starts selected level.

IV.Side Panel

The side panel is found at the right side of the screen and it is divided into two halves. The upper half is the ‘view area’ and the lower half is the ‘edit area’.

- View area

There are three main buttons in the view area. In the Static view we have:

FIGURE 3-4 : THE STATIC VIEW.

1. Information of the attractor the particular state is contained, and whether the state is in the cycle, or in the rest of the sequence.
2. A state selector with the next button that will evolve the net to its next state (in Hex).
3. The history of the states, where with yellow is depicted the next state, in blue the current and in purple all the past states.

Next we have the ‘Limit’ view which will show us information about the attractors and the cycles.

FIGURE 3-5 : THE LIMIT VIEW.

1. Information about the length of the attractor and the cycle, along with the energies associated.
2. The attractor selector (in Hex).
3. The sequence of the attractor. In yellow the states of the cycle, and in blue the rest of the sequence.

Finally, there is the history view, where you can:

1. View the history of the moves you and the AI have made.
2. View the solution.
3. Skip turn if in AIOP game.

- Edit area

In the Edit Area is where most of the editing will happen. There are two main view and are accessible once you select a net object.

For the nodes we have:

1. A slider to modify the threshold of the node. Remember that sum of excitation in the input of the nodes has to be >= of the threshold for the node to get excited.
2. Lists of the IDs of the input and the output edges.
3. Buttons to reset the state of your modifications, remove the node, apply the changes and add a new Node.
4. Information about the current restrictions.

FIGURE 3-6 : NODE EDIT VIEW.

In case an edge is selected:

1. A button to modify the type of the edge.
2. Two selectors for the IDs of the in and out nodes.
3. Buttons as above.
4. Information space for the current restrictions.

FIGURE 3-7 : EDGE EDIT VIEW.

V.Net Area

The net area is the space where the Net appears. There are four operations that tapping or clicking can do.

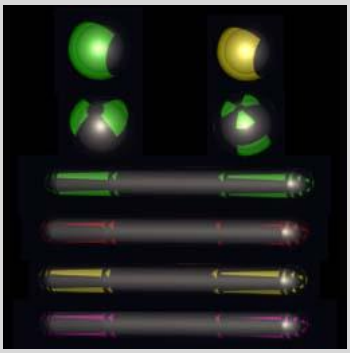
- Operations.

- Rotation: Hold down finger or mouse and move.
- Zooming in/out. Use to fingers on a portable device, or right-click and drag on desktops.
- Add a new node by tapping on an empty space. This will work only if adding a new node is permitted at the current state.
- Add a new edge. Select the starting node. Hold finger on starting node, drag until the ending node is selected. Release finger.

- Object colors and motion

Nodes can have two colors. Green means there is no restriction. Yellow means the node is not removable. In Figure 3-8 on the second line are depicted nodes that have different thresholds (2 and 3).

Edges can have four different colors. Precisely:

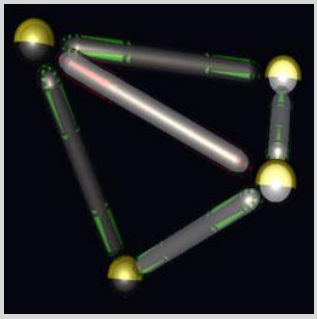


1. Green: excitatory and removable.
2. Red: Inhibitory and removable.
3. Yellow: excitatory not removable.
4. Purple: Inhibitory not removable.

FIGURE 3-8 : NET OBJECT COLORS.

Finally, for both nodes and edges, if the texture is still, it means that node threshold or edge type is not editable.

- Net Highlight



As you can see in Figure 3-9 and Figure 3-10 there are cases where different visual effects can take place. In the first case excited nodes and edges in the state selected, take a brighter inner color.

At the second case nodes and edges that take part in the visual attractor, take a purple hue, that has transparency linked to the energy of the cycle.

FIGURE 3-9 : STATE OBJECTS HIGHLIGHTED.

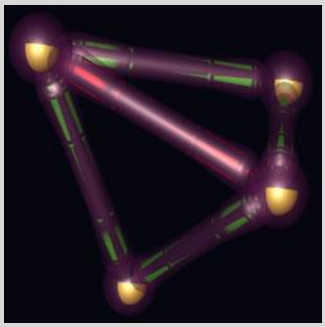


FIGURE 3-10 : VISUAL ATTRACTOR HIGHLIGHTED.

4.The AI

I decided to include an AI opponent to the game, because I was looking for an occasion to use deep neural network in one of my projects. As it will be shown in this chapter, this is not the best suited application that one could think of, but my main purpose was to show that some fundamental techniques used in reinforcement learning are easily adapted and reproduced [1].

I. Reinforcement Learning and Q-Learning

Reinforcement learning, is one of the few available ways to train a neural network, without providing labeled data to the network (unsupervised). The main idea behind RL is that an agent living in an environment will have internal states, take actions, receive rewards from the environment, and finally alter its internal state in order to maximize its future rewards. If the system's ability to switch state is described by a probability, then the system is called a 'Markov Decision Process' and is theoretically very well described in the realm of probabilities theory. But in our case, what we would really want to do is to be able to force some probabilities to one and others to zero, so as to maximize the future sum of rewards.

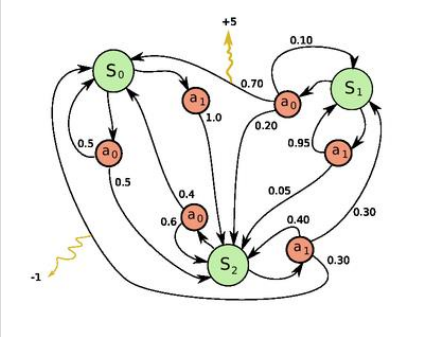


FIGURE 4-1 : MARKOV DECISION PROCESS (FROM WIKIPEDIA)

The most popular method to solve this problem, is through solving the Bellman equation:

$$Q(x_0) = max_\alpha\{R(x_0, \alpha_0) + \beta Q(x_1)\}$$

Where $Q(x_0)$ is the maximum reward obtainable given that we are in state x_0, $R(x_0, \alpha_0)$ is the reward we will take from just the action α_0 taken from state x_0 and $0 < \beta \leq 1$ is the discount factor that reduces the importance of rewards very far in the future, in order to avoid divergence. This is a very intuitive equation that breaks the problem into two sub problems and can be solved iteratively: The maximum reward obtainable starting from state x_0 is equal to the best sums of the rewards available from current state, plus the maximum obtainable reward obtainable from the state that this action will take us to. Notice also that having the max operator around the whole summation and not just around $R(x_0, \alpha_0)$, lets the system take a possibly not so good action now, that will eventually provide better rewards in the future.

When the bellman equation is approximated by a neural network, then the technique is called Q-learning and is having a lot of success and attention lately. In order to implement Q-learning, all we need is a model of the system to simulate the actions onto, a reward definition for each action-current_state pair, and a neural network. For each Training epoch, $R(x_0, \alpha_0)$ is given by the model, and $Q(x_1)$ is given by the Neural Network. Of course initial ignorance in the network, will start giving wrong predictions of $Q(x_1)$, but

(1) V. Mnih, K. Kavukcuoglu, D. Silver. 2013 Playing Atari with Deep Reinforcement Learning
M. Lai. 2015 Giraffe: Using Deep Reinforcement Learning to Play Chess

that will eventually tend to more sensible results, as the training evolves and more examples get into the pool of action-current_state pairs.

II.The implementation

In our case, I have implemented the neural network with the CNTK framework, and the training was done with the python interface that is provided. It has to be noted at this point that CNTK, in order to evaluate the networks from C# provides a .NET library that is compiled on .NET framework 4.5. Those who are familiar with Unity, will know that at the current date unity does only support up to .NET3.5. Fortunately, there is an experimental version of Unity (Unity5.6.0b5_Mono_Upgrade) that permits such libraries to be included, but is very unstable. At the end I have been able to include CNTK only in the standalone version and so the Android version will not have the AI, until a stable version of Unity supporting .NET4.5 will be out.

Hereafter is a list of problems faced and choices made during the development.

- ### Constructor vs Destroyer

Choosing the goal for the game has not been very difficult in the sense that there weren't many choices. The two opponents had to play on the same network, as the actions of one player had to interfere with the decisions of the other, because if not, the AI would have a deterministic behavior. Then one had to choose the property which the two players will have to control, and choosing to modify the number attractors seemed the most basic and maybe easier for the NN to learn. Thus the player can choose to create more attractors (Constructor) or try to keep them at the minimum of one (Destroyer).

- ### Biased game

Having set the game goal, the next obvious observation was that the game was starting to be very biased by definition. It is always much easier to destroy than create, and this game was not an exception. There are tens of more moves that reduce the number of attractors, than the moves that increase them. This was immediately obvious while training, and we will see next some actions that I took in order to mitigate this effect. But still after those modifications, playing as a constructor was very penalizing, so much that if the destroyer chose moves at random, would still probably win the game. To show how 'unfair' it is to play as a constructor, when the player choses to be a constructor, then the opponent will not choose his moves from a NN but completely randomly. The NN is used only in the cases where the AI plays as a constructor.

Another modification is done in order to help the constructor, is to stop the game early by reducing the actions available. More precisely, each time a node threshold is modified, then the edges touching than node are blocked and cannot be modified until the end of the game. Same rules apply for the nodes touching an edge on which we have just took an action. Finally, the total number of moves is also reduced to 8.

- ### Variable actions Count

If the number of actions is not very large and variable, then there is a clever simplification that can be made to the net. That is one can give to the net as input only the

current state, and have the net predict as many $Q_\alpha(x_0)$ as there are α. This way the evaluation of all the actions is done at a single iteration, and the network and its training is much simplified, because it does not have the codification of the action as input to the Net. Unfortunately, this is not the case for this game, because the available actions in each move is highly dependent on the state the game is in. But still I had to find a way to simplify the inputs to the NN so what I did was to try to include in the codification of the state the codification of the actions too. That is, I codified the state of the game with zeros and ones, and the actions with -1. i.e. if the first 7 bits of the input are these: {0,0,0,1,0,0,0}, it means that the node with id=1 has a threshold of 4 prior and after the action, while this code: {0,-1,0,1,0,0,0} means that the action that is taken is to modify the threshold of the node from 4 to 2. Similar rules apply for the edges actions.

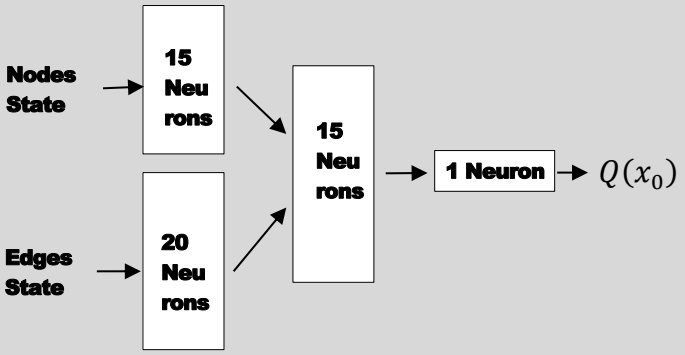


FIGURE 4-2 : NEURAL NETWORK USED FOR THE PROJECT

- ## Variable inputs – Many nets

Having defined the inputs as described above, leads to a variable need of inputs (76 for 4 node CogniChessNet 217 for a 7 node CCNet) which also might be very similar with very different results. i.e. the actions of changing the threshold of two different nodes starting from the same CCNet have all the inputs equal except 2 and might also have drastically different results. I concluded that despite what happens usually with NNs, in my case overfitting was a friend. In order also to accommodate overfitting, I had to train different NNs for each level in the game.

I did find a way to have a generic input definition, but the results were not as good as having the nets dedicated.

- ## Few winning samples, had to discard many zero rewards

When it comes to overfitting your NN, then you have to present to the input all the cases that you want to be overfitted. The problem was, that having so few winning-constructing moves, they get lost through the actions that did not give any reward or that gave negative rewards. The actual winning actions that I wanted the net to remember, where overwhelmed by the tons of actions that were very similar, but did not bring positive reward. Two actions were made to overcome this problem. First a rewards definition was revisited in order to give a slight positive reward even to actions that were taking the CCNet one step closer to a good result, and secondly, I used a pool of mini-batches that had a given percentage of zero, negative and positive rewards. In my case I used mini-batched that were forced to contain 40% of zero rewards, and 30% for negative and positive.

- Low processing power

Finally, it has to be made clear that I did not have the means to make proper training sessions as this project would deserve. I only had my personal laptop to work on, and I did not let each training session overpass the 10 hours or the 60000 epochs. There is evidence to suggest that with better tools this AI could have been a lot better, and even if at a first glance might appear that this was not a big success, from my point of view, it shows that Q-learning is very flexible and can be applied to many different study cases.