# Dots-and-boxes

In a grid of N x N dots we have a total of NT = 2N(N – 1) edges. (So 40 *for N = 5 and 60 for N = 6)*

Each edge can either be drawn or remain empty. This gives us a total of 2^(NT) different states.

In each state, we have the option of selecting an action to illustrate some (so far empty) edge.

The total space of (state, action) pairs has |P| elements.



This is an exponential function, which causes the total state-action pair space to become very large, even for relatively small values. *(e.g.|P| > 227 for n = 5 and |P| > 265 for n = 6)*

## 2.1 Different approaches:

We can mainly divide our approaches in two categories.

Based on our experience in creating this project, we could divide our methods into two types. (If someone shows me a new game and wants me to learn how to play the game, I have two options to do it)

1. Het type denken (eerst, daarna, dan speel ik) - ik zit op het plan en begin het op de een of andere manier te analyseren, waarschijnlijk het beste van het einde en probeer te onthouden welke posities goed voor me zijn en welke niet en wanneer iemand het wil spelen, Ik probeer mezelf naar posities te slepen die slecht zijn voor de tegenstander.
2. Type of player (first I play, then I think) - I play a few experiments games in which I play more randomly and try something from them EIT. Gradually, as I learn my knowledge, I apply when playing other games.

Of course there is a third type - some combination of these two.

Let's look at the known methods: value iteration, Q-learning.

* + 1. Value iteration: (twoord value iteration got me confused)

Value iteration is definitely thought. However, it is not considered (n> 4), because repeatedly going through all the states would take too long and it is virtually impossible to keep a complete table of values for all pairs (status, action). One way to create a decent player is, for example, to use value iteration for the last K strokes and for the preceding ones use a reasonable heuristics.

* + 1. Q-Learning

Q-learning also has a similar negative. We attempted to model the Q function using various ML methods. The biggest problem is choosing the right method. Most methods are designed to prevent what's a bit of a problem in our case. One obvious, forced, good move around the very evil may seem in the data exactly as an outlier, respectively. error and most methods do not. Thus, only neural networks and decision trees remain, since they have relatively large freedom of fitting.

‘’’Q-learning also has a similar negative. We tried to model the Q function using various ML methods. The biggest problem is the choice of the right method. Most methods are designed to prevent overwhelming problems. One obvious, forced, good move around the very evil may seem in the data exactly as an outlier, respectively. error and most methods do not. Thus, only neural networks and decision trees remain, because they bring about a fairly high degree of honesty.’’’

The second major problem is training. As input for Q, we used a binary vector representing the current play area (1 if the edge is 0 if it is not). Player type trapping is a problem, because few models support online learning (gradual learning with fewer pieces of data).

And when trying to access this approach, it was very often that the values jumped too much. Only in very small neural networks it looked good from the beginning (managed to get at least some good starting points), but after a lot of games, the save of these values changed and therefore all the other updates were almost nonsense. I did not succeed in achieving this result. It might be interesting to add some other than the vector representing the playing area predicted variables. Or try out another way to update Q.

Typewriter training would take too long, and especially to generate all training examples at once that we want to fit is almost impossible. It is quite reasonable to think of each "layer" of Qm, where under the layer we think of all the areas of the area, where the edges are left unbroken.

So if we proceed from the end, we know exactly Q0, Q1, Q2, etc. This is what we used to train the optimal game of the last K strokes, it would be best to use the decision tree for each layer. However, Qm would still have to identify and then fit the decision tree, as long as they do not support online learning. So the only advantage would be that the trained tree would take less memory.

And when you try this approach has often happened to have jumped in value. Only one of those small neural networks looked good at the beginning (at least some of the correct starting values ​​were trained), but after many games many of these values ​​have changed, and so all other updates have been almost overwhelming. I did not succeed in achieving this result. It might be interesting to add some other preset variables except for the vector representing the playing area. Or, try out another way of updating Q. It would take too long to train mind-setter, and especially to generate all the training examples that we want to make is almost impossible. It seems to be quite sensible to train each "layer" Qm, where under the layer we think all the states of the area where the edges are left unbroken. So if we proceed from the end, we know exactly Q0, Q1, Q2, etc. This is what we used to draw the optimal last game play, for which it would be best to use the decision tree for each layer. However, Qm values ​​would have to be ascertained first and then deciding on the decision tree, as long as they do not support online learning. So the only advantage would be that the trained tree would take less memory.

## 2.2 Our implementation:

The first question is how to reward it, as it is teaching it with reward and punishment. We can reward each point earned or reward the game at the end of the win (or punish for a loss).

2.2.1 Reward for every point

Under conditions of the game we will understand the state of the court (in using or without of who is on the move and what the current score), because only that would affect how the move will make optimal players. It is obvious that in every position is a best move (ie. A set of best strokes) and our goal is just looks for these moves.

The storage value (whether Q-function, and values ​​in the iteration value) for a pair (state, action) a little easier. If we used a value-iteration or Q-learning as a right of the enclosure, so do not use speci ka that this game offers.

While we know the exact state in which we move the game after our move, we can precisely determine our reward. If we earned a single point for each point we have to have a discount factor = 1. This discount factor does not make sense now, because the game is still so long, and the score we do not degrade with time, and unnecessarily put emphasis on moves. which we get the box earlier, instead of pulling the corners, we get more boxes, but later. Imagine such an interpretation of values. We'll remember the status with

Q(s) = [How many points is it possible to get the player to pull from this turn to the end if we and the opponent are playing the best as we can]

Now you see, because we do not have to remember the value especially for us in the state of s and especially for the opponents. So if the player on the move knows how to get Q (s) points, the other can get all the others that have not been won.

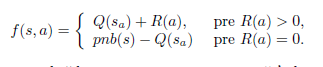
Such a Q function must satisfy certain relationships. If I'm in S-state, I'm doing Action A and the area is in SA state, there are two possibilities:

1. I Gained R (a)> 0 points, so I will pull myself again from the state.

Then we want Q (sa) + R (a) to be the highest (to get as many points as possible).

1. I get R (a) = 0 points so I will pull the opponent in the SA state. Then I want Q (SA) to be as low as possible (to get the opponent to a position where he gets the least points). Or so from my point of view, if he can get the maximum Q (s) points, I know how to get it is at least all the others, so I want to maximize pnb (s) - Q (sa) where pnb (s): = [# unobtained points]

Let us denote f (s; a) a function that combines these two possibilities and returns the number of points I know to get when I'm in S state and I do action A.

(PRE = for)

The best action abest in each state S can then determine how

Abest = argmaxa€A(s) f(s,a)

This relationship also offers a way to determine Q(s), if we know Q(sa) for all a € A(s). So if we are in a state we can get Q(a) points and make the optimal move, so we know with him in the likeness of the game exactly Q(s) points, therefore



Our new Q we will update as follows:



The advantage of this representation of Q is compactness - lower memory requirements, faster training, another advantage is that if we train it while playing, the player will never be swayed. The more games he plays, the better he'll be, even if he plays against bad opponents, so the learning rate can be a sliding one. The disadvantage is that we now have to value the value of f. But it's a good trade-off, because the time for calculating f for voters will not grow, unlike the number of shares for which we had to remember these values.

**Storing Q**. We will store the Q values in the dictum sheet - m-th dict Q [m] will be applied to the mth layer, i. the playing area of the game area, where it is lacking to complete the play. And its columns will be the numbers of the individual plots. we get the number of the surface (binary) so that we present each edge as a bit and if there is a number 1 if not, there is 0 (in a fixed order). Then it is easy to change area numbers when adding one edge (what we do at each stroke), just add the number of that edge.

2.2.2 Reward at the end

In order to be rewarded at the end, we need to include (in addition to the status of the playing area) some indication of the current score. It could have a good effect on the outcome, but training would take a long time, and especially a trained player might not be consistent. In the same state with different current scores, he could think of another action as optimal (for example, when he knows he's lost, so he does not care). And while my computer memory and time are very limited, I am confined to the higher payout described during the game. (of course, there is always the possibility that there is something about which we did not write and can be rewarded at the end of the better)

2.2.3 Training

In mind-based training, the procedure is clear. We train Qm by layers, gradually from the end of the game. This way we can only pass all our positions once. We have created a variety of opponents for our type of training, against which we have simulated training (and testing) games.

* **random move** always creates a random free edge (evenly random)
* **first available move**, creates the first leading edge according to a fixed order (determined distance from the lattice point (0,0)). Retrieved from [2]
* **always4never3**, always completes a stack whenever it can (always4), if it is not possible to leave the three-edged player anywhere (to prevent the opponent from completing it), otherwise he plays randomly. Retrieved from [1]
* **bot2** (k) - the last moves are optimal, the game always plays with always4never3

As well as being able to play against these players (and their casual combinations), it has been possible to play random strokes (exploration of new positions) and turn moves according to the current Q (updating the best strategy according to new knowledge).

**Updating:**

Q took place after each game. In this case, the information is always shifted by one layer. However, if we remember all the moves in the game, we can move all the new information later from the end of the game to the beginning. (It's like when a man re-analyzes the game played at the end of the game)

Specifically, if I update during the game, I will update Q [m] according to current Q [m-1], I will update Q [m-1], Q [m-2] etc downwards. this update is no longer moving. It is reasonable to check at the end of the game, or the Q [m] thus changed does not change Q [m + 1], even upwards.

**Initialization**

An interesting question to think about is how to set values for

Q for the states we did not see? The reasonable value is not 0, because if we do not get a point at that turn, it seems as if it was extremely advantageous to leave this position to the opponent.

Then we will visit it, we will find an estimate of its value (often larger than 0), and in the future we will probably never leave this position anymore, because we have a choice of many with a value of 0.

Reasonable may be an estimate of the mean value - the average (eg if 5 points remain at the end of the game, we assume that not everyone knows Q [m] is 2.5, or if we already know some values we can choose their average) estimate, the crowd will collect all the interesting (higher and lower) values. If the average has been overstated, it will make us more experienced we do not know these values until we make a better estimate of our average.

# Conclusie: