Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Artifictial neural network as an opponent in Quoridor

Bachelor Thesis

2016

Michal Hoľa

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Artifictial neural network as an opponent in Quoridor

Bachelor Thesis

Study programme: Applied Informatics
Study subject: 9.2.9 Applied Informatics
Department: Department of Applied Informatics
Advisor: Mgr. Peter Gergeľ

Bratislava, 2016

Michal Hoľa

Declaration of Authorship

I do solemnly declare that I have written the presented thesis by myself under careful supervision of my thesis advisor without undue help from a second person other than that specified.

. . . .

Acknowledgement

I would like to thank my advisor Mgr. Peter Gergeľ for suggestions, help, guidance and friendly approach.

Abstract

...absctract text here...

Contents

Chapter 1

Introduction

Since it is in our nature to play games, we seek for better opponents to improve ourselves. There are variety of games where we can enhance our mental abilities. However, there were found optimal strategies for some of them, which may sometimes seem eigther less challenging or even less interresting. Nevertheless, game Quoridor is not among these yet. Considering it has been invented relatively recently, attempts realized to create a computer agent have been scarce. Additionaly, there has been no success with the use of artifitial neural networks. And this is what within the scope of this thesis I will try to accomplish.

To train neural network, I will create opponent with simple heuristics. Moreover, I will combine Q-learning with networks ability to generalize to estimate best action.

Chapter 2

Background

In this chapter, there will be introduced Python programming language and different existing approaches in solving board games with agent based model.

2.1 Python

Python is widely used and popular programming language. It is an interpreted, high-level, dynamic programming language supporting multiple programming paradigms, including functional, object-oriented and procedural style. It has been designed with regard to readability, easy adoption for novices and scalability of programs. Python offers a lot of utilities as part of its comprehensive standard library. It is possible to work in Python under multiple operating systems and it is often standard component among many linux distributions. Main reasons why I chose Python was that I had four years of experience with it and for its maintainability.

2.2 Agent based model, Markov decision process and Game theory

Agent based model (ABM) is a paradigm in modeling systems comprised of autonomous agents. Also, it is a class of computational models for simulations of the actions and interactions between agents. Althought, there are multiple definitions of what is considered agent [1], often, agent is an identifiable component with reaction rules able to make independent decisions (or behaviours) and sometimes with ability to learn and adapt to the environment.

Markov decision process (MDP) is a mathematical model for describing stochastic discrete time decision processes. Important property that the proccess must have is a memorlylessness, which means that the next state depends only on current state and not on any other preceeding states.

Game theory studies conflict or cooperation between rational decision makers and also different strategies for given problems. Knowledge from computational complexity theory is used to determine or estimate game complexity. Here, MDP is often used to describe such rational decision makers.

When trying to find the optimal strategy where due to the large number of states in a game it is not feasible to use brute-force, there are often used:

- Heuristic methods - A-star, Alpha-beta pruning
- Reinforcement learning (RL) - Monte-carlo algorithms, Temporal difference methods, Q-learning
- Evolutionary methods - genetic algorithm
- Artificial neural networks combined with RL methods

2.3 Minimax

The principle of the algorithm is to find the best action for an agent by traversing a game tree while minimizing maximum possible losses. The requirement for the algorithm to work is existance of a function capable to evaluate arbitrary position in the game. Also, it is assumed that there are no infinite sequences of positions allowed.

Algorithm walks througt all the possible moves, performs evaluation of subsequent positions and picks up the action bringing the most advantageous position. Evaluation is done by static evaluation funciton or the same algorithm is executed recursively for the opponent. Often, recursion has limited depth to ensure the algorithm ends in a reasonable time.

2.4 Alpha-beta pruning

Alpha-beta pruning is enhancement of minimax algorithm. On average, it speeds up the minimax algorithm by eliminating branches of a game tree which will be certainly left off from choosing.

2.5 Reinforcement Learning

TODO

2.6 Q-Learning

Q-Learning is reinforcement learning method. It learns an action-value (q-value) function for any finite Markov decision process used to find optimal policy for the agent. After it has learned the q-value function, it can follow the optimal strategy by choosing best q-value. ?citation? Algorithm is value iteration update for every observed state ?citation?:

$$Q_{t+1} \leftarrow Q_t(s_t, a_t) + \alpha \cdot (R_{t+1} + \gamma \cdot \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{2.1}$$

where $Q_t(s_t, a_t)$ is old value, α is learning rate, R_{t+1} is reward after performing action a_t in state s_t, γ is discount factor and $\max_a Q_t(s_{t+1}, a)$ is maximum optimal future value.

2.7 Evolution methods and genetic algorithms

TODO

2.8 Perceptron

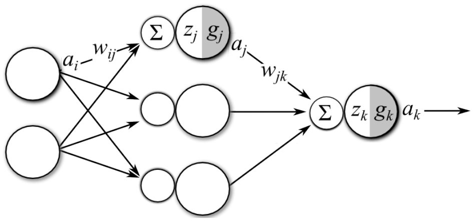


fig. 2.1: ANN

Artifitial neural network (ANN) is a family of models inspired by biological neural networks used in computer science to approximate functions with large number of inputs. Generally, artifitial neural network is presented as a system of interconnected neurons exchanging messages between each other. These connections have weights that can be adjusted based on experience which makes the network capable of learning.

Perceptron is an algorithm for supervised learning of binary classifiers where one neuron has multiple weighted inputs and single output. Single perceptron can learn to decide between two linearly separable classes. Multiple perceptrons in multiple layers (MLP) use arbitrary activation function which makes it able to perform classification or regression based on the activation function chosen.

2.9 Perceptron estimating Q-values

$$\hat{Q}^{new} \leftarrow r_t + \gamma \max_a \hat{Q}(s_{t+1}, a) \tag{2.2}$$

Main advantage of ANN is its ability to generalize. This is has been used in game othello [2] with combining Q-learning so that network has learned to estimate q-values for each state. Learning rate α has not been directly used in iteration update (2.2) since ANN has also learning rate so it can be adjusted there.

Chapter 3

Quoridor

3.1 History

Game Quoridor was invented by Mirko Marchesi based on game Blockade and has been published and sold (fig. 3.1) by french company Gigamic Games in 1997. Since it has been developed relatively early compared to other games such as Chess or Go, there have been only few attempts of analysing the game and creating game agents.

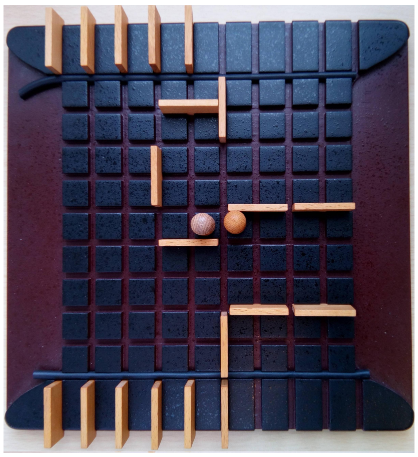


fig. 3.1: real board

3.2 Rules

Quoridor is abstract board strategy game for 2 or 4 players with size of 9x9 (81) squares. This thesis covers 2 player version of the game.

Each player starts (fig. 3.2) with a single pawn in the center of the edge on the opposite side as the opponent. The goal for each player is to reach the opposite edge.

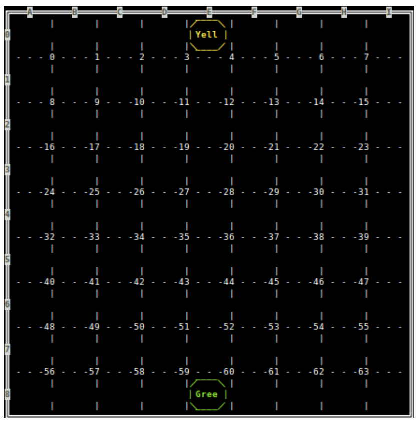


fig. 3.2: game start

Player also starts with 10 walls (fences) in the stock. Walls are two space wide and can be placed in the groove that runs between the spaces. Placed wall blocks pawns paths forcing them to go arount it. Walls once placed can not be moved nor removed. Wall can not be placed to the position already occupied or crossing by other wall. Also, wall can not cut off the only remaining path of any pawn to his goal.

When player is on turn, he must place wall, if he has left some, or move his pawn to adjacent (not diagonal and unoccupied) space. If opponent's pawn stands on an adjacent space, current player can jump with his pawn to all the places where the opponent pawn can move.

3.3 State Complexity

Estimated game state complexity was $3.9905 \cdot 10^{42}$ [3] (eq. 3.1), however, this is very rough estimate, since it includes many states multiple times where it counts with permutations instead of combinations of walls.

$$\begin{aligned} S_p &= 81 \cdot 80 = 6480 \\ S_f &= \sum_{i=0}^{20} \prod_{j=0}^{i} (128 - 4i) = 6.1582 \cdot 10^{38} \\ S &= S_p \cdot S_f = \mathbf{3.9905 \cdot 10^{42}} \end{aligned} \tag{3.1}$$

My approach (eq. 3.2) with estimating state complexity will be similar. I will estimate maximum states of this game, which will include impossible states such as:

- walls crossing each other
- pawns not having the path to the winning position
- pawn in the winning position where it could not end due to walls

Moreover, this estimate will differ between states when different player is on the move, and also, when there is different number of walls in players stocks. Both of these could make the game very different in the outcome.

$$\begin{aligned} S_p &= 81 \cdot 80 - 9 \cdot 9 = 6399 \\ f(i) &= \begin{cases} i + 1 & \text{if } i <= 10 \\ 21 - i & \text{if } i > 10 \end{cases} \\ S_f &= \sum_{i=0}^{20} f(i) \binom{128}{i} = 1.7796 \cdot 10^{23} \\ S &= 2 \cdot S_p \cdot S_f = \mathbf{2.2775 \cdot 10^{27}} \end{aligned} \tag{3.2}$$

S_p was corrected to not include both pawns in the winning positions. $f(i)$ stands for different wall counts in the stocks. $\binom{128}{i}$ are all (valid and invalid) combinations of walls. 2 in $2 \cdot S_p \cdot S_f$ represents different players on turn.

The result is maximum number of possible states, which is significantly less then former estimate. However, even if I could evaluate 10^6 states in one second, it would take around $7.2172 \cdot 10^{13}$ years to evaluate this many states. Also, lets assume it takes on average 50B of memory per state. Then I would need approximately $1.1387 \cdot 10^{29}$ bytes of memory to store all states in the computer. This is why simple Q-learning is not feasible.

It is worth to note, that if invalid wall combinations with walls crossing could be removed from $\binom{128}{i}$, this estimate would be very precise.

3.4 Game Tree Complexity

Average bronging factor of the game has been experimentaly measured to be 60.4 and average game length to be 91.1 [4]. Mertens [3] used this to compute game-tree size G:

$$G = 60.4^{91.1} = 1.7884 \cdot 10^{162} \tag{3.3}$$

To create my own estimate of branching factor, I will estimate number of states where decision can be made and also number of different decisions possible. Branching factor will be simply division of these.

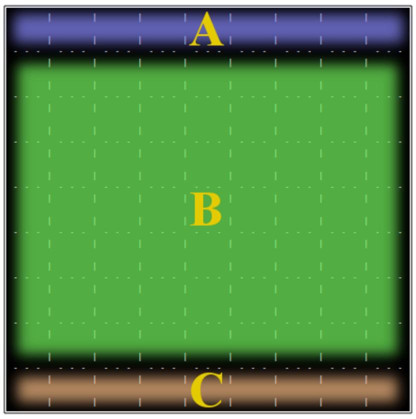


fig. 3.3: board areas

To count every possible pawn move, board is divided into three areas (fig. 3.3), where A and C are positions from the first and last row respectively and B are positions from 7 rows from the middle. So, position AC means, first pawn is in the first row and second pawn is in the last row or BB means, both pawns are in the middle 7 rows. Then, for each pattern (fig. 3.4), possible positions for making decision and possible decisions are counted.

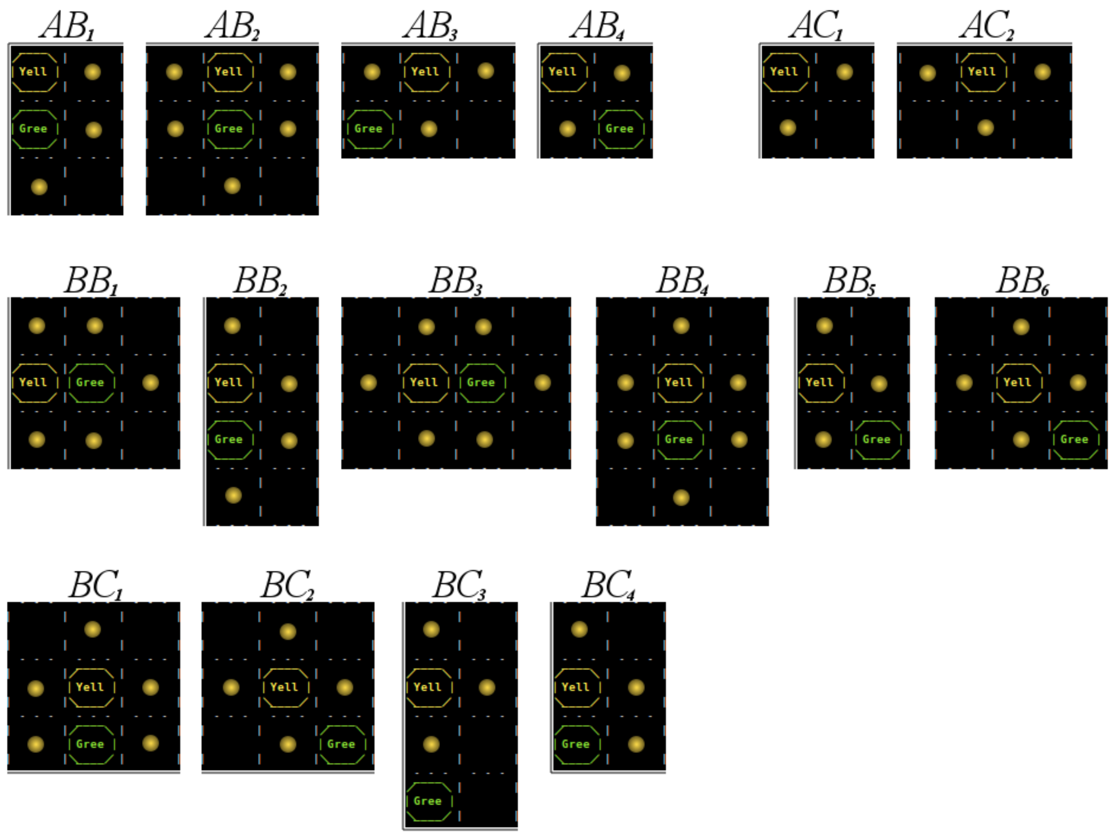


fig. 3.4: decision patterns

In total, there are 3168 different positions where pawns can make decission and from that, there are 11309 different decision that pawns can make (see tab. 3.1).

	Positions	Decisions
AB_1	2	$2{\cdot}3 = 6$
AB_2	7	$7{\cdot}5 = 35$
AB_3	$7{\cdot}62 = 434$	$434{\cdot}3 = 1302$
AB_4	$2{\cdot}62 = 124$	$124{\cdot}2 = 248$
BC_1	7	$7{\cdot}5 = 35$
BC_2	$42{\cdot}9{+}7{\cdot}8$ $= 434$	$434{\cdot}4 = 1736$
BC_3	$6{\cdot}9{\cdot}2{+}2{\cdot}8$ $= 124$	$124{\cdot}3 = 372$
BC_4	2	$2{\cdot}2 = 4$

	Positions	Decisions
AC_1	$2{\cdot}9 = 18$	$18{\cdot}2 = 36$
AC_2	$7{\cdot}9 = 63$	$63{\cdot}3 = 189$
BB_1	$7{\cdot}2{\cdot}2 = 28$	$28{\cdot}5 = 140$
BB_2	$6{\cdot}2{\cdot}2 = 24$	$24{\cdot}4 = 96$
BB_3	$6{\cdot}2{\cdot}7 = 84$	$84{\cdot}6 = 504$
BB_4	$6{\cdot}2{\cdot}7 = 84$	$84{\cdot}6 = 504$
BB_5	$2{\cdot}2{\cdot}60{+}5{\cdot}2{\cdot}59$ $= 830$	$830{\cdot}3 = 2490$
BB_6	$\binom{7\cdot 9}{2} - \sum_{i<6} BB_i$ $= 903$	$903{\cdot}4 = 3612$

tab. 3.1: positions and decisions

Next, maximum numbers of possible wall placement N relative to walls already placed i are shown in table 3.2:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	128	125	123	120	118	115	113	111	109	107	105	103	101	98

i	14	15	16	17	18	19	20
N	96	94	92	90	88	86	0

tab. 3.2: wall placement possibilites

Optimal pattern for taking as least as possible from future wall placement possibilities is shown in figure 3.5. Naturally, placement starts in the corner and the following walls are placed adjacent to each other.

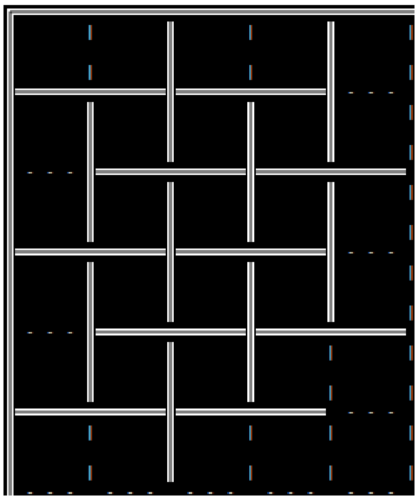


fig. 3.5: min. tesselation example

Now, we can estimate all diferent states S_d for making decision (eq. 3.5), where $f(i)$ is the same function for stock factors from equations 3.2, $\binom{128}{i}$ is top estimate of different wall placement combinations.

Number of different decisions possible D are sum of all possible movemets and all possible wall placements. For wall placemnts (eq. 3.4), there are different stock factors $g(i)$, since state where player moving has no walls in stock cannot place the wall.

$$g(i) = \begin{cases} i+1 & \text{if } i <= 9 \\ 20-i & \text{if } i > 9 \end{cases} \tag{3.4}$$

$$\begin{aligned} S_d &= 3168\sum_{i=0}^{20}\binom{128}{i}f(i) = 5.6379{\cdot}10^{26} \\ D &= \sum_{i=0}^{20}\left[11309f(i)\binom{128}{i} + 3168g(i)N(i))\binom{128}{i}\right] = 1.0766{\cdot}10^{28} \end{aligned} \tag{3.5}$$

However, S_d and D include states where there are walls crossing each other and pawns not having path to the winning position. Moreover, D does not count with number of movement possibilities taken by placed walls.

$$b = \frac{D}{S_d} = \frac{1.0766{\cdot}10^{28}}{5.6379{\cdot}10^{26}} = 19.0974 \tag{3.6}$$

The resulting estimate of branching factor b is significantly lower than the former estimate. But, it may be biased a little towards smaller number. It would be caused by more combinations in situations with all walls placed having more impossible possitions than the other.

Game tree complexity C calculated with new branching factor estimate and with former average game length:

$$C = 19.0974^{91.1} = \mathbf{4.9758{\cdot}10^{116}} \tag{3.7}$$

Chapter 4

Implementation

4.1 Game

Game class is often implemented as a set of tools operating on provided state. With this approach, tests can be easily designed to verify functionality. In many cases, following methods or attributes are available:

- initial_state - initial state of the game
- is_terminal - whether the given state is terminal
- valid - validates given state or action on the given state
- actions - list of all possible actions in the given state
- execute_move - returns new state after executing given action
- utility - value representing how the given state is beneficial for given player

I consider following methods also important in Quoridor, so these are present in my implementation:

- undo - previous state before provided action and state
- shortest_path - shortest path to goal row for the given player
- path_blockers - set of wall positions that could block the given path
- crossing_actions - set of invalid actions that would cross already placed walls

4.2 State

State should be the smallest possible representation of the game state where it can be easily distinguished between any two different states. Mathematicaly, state S can be defined as:

$$S = (c, p_y, p_g, s_y, s_g, w) \tag{4.1}$$

Here, $c \in \{0, 1\}$ represents color of the player that should move, $p_y, p_g \in \{0, 1, ..., 80\}$ are pawns positions, $s_y, s_g \in \{0, 1, ..., 10\}$ are counts of walls in stocks, and set of positions of placed walls are $w = \{\, a_i \,|\, a_i \in \{0, 1, ..., 127\}, i \in \{0, 1, ..., 20\}\}$. One such state in python would be:

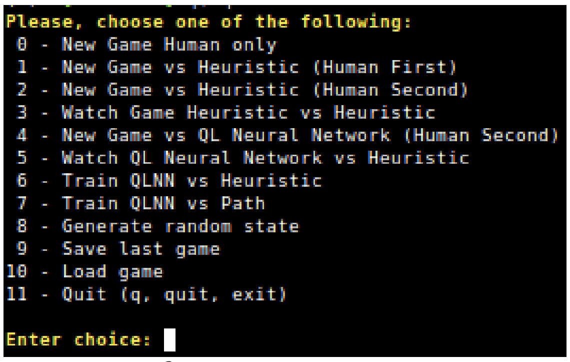
```
state = (0, 51, 70, 5, 9, frozenset([32, 116, 119, 59, 12, 31]))
```

However, when providing state information to ANN, it always has to be a vector of same length, so placed walls are represented by 128 zeroes or ones. Then, the result is vector with 133 values.

4.3 Context

For speeding up validation of the moves for agents, it turned out to be faster to keep in memory shortest paths and invalid actions such as crossing walls. `QuoridorContext` class serves this purpose. Also, this speeded up the decision process of heuristic and path agents.

4.4 Console Game



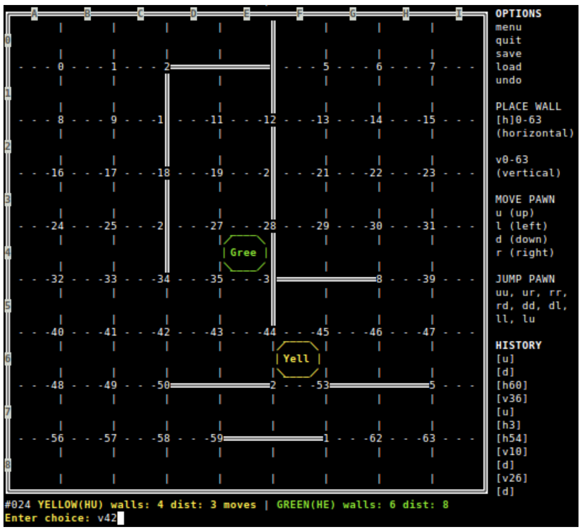
```
Please, choose one of the following:
 0 - New Game Human only
 1 - New Game vs Heuristic (Human First)
 2 - New Game vs Heuristic (Human Second)
 3 - Watch Game Heuristic vs Heuristic
 4 - New Game vs QL Neural Network (Human Second)
 5 - Watch QL Neural Network vs Heuristic
 6 - Train QLNN vs Heuristic
 7 - Train QLNN vs Path
 8 - Generate random state
 9 - Save last game
10 - Load game
11 - Quit (q, quit, exit)

Enter choice: 
```

fig. 4.1: main menu

Main purpose of my work was on implementation of the agent utilizing abilities of the ANN, so the game is played and displayed in the linux terminal (console), and so, all interactions are performed through command prompt. To display the game properly in the terminal, it must be at least 86 characters wide.

In the main menu (fig. 4.1), user can choose to play a game against `HeuristicPlayer` or `QLNNPlayer` and also, to watch a game played eighter by `HeuristicPlayer` versus himself or against `QLNNPlayer`. Also, loading a game and saving last played game is possible.



```
OPTIONS
menu
quit
save
load
undo

PLACE WALL
[h]0-63
(horizontal)

v0-63
(vertical)

MOVE PAWN
u (up)
l (left)
d (down)
r (right)

JUMP PAWN
uu, ur, rr,
rd, dd, dl,
ll, lu

HISTORY
[u]
[d]
[h60]
[v36]
[u]
[h3]
[h54]
[v10]
[d]
[v26]
[d]

#024 YELLOW(HU) walls: 4 dist: 3 moves | GREEN(HE) walls: 6 dist: 8
Enter choice: v42
```

fig. 4.2: game running

During the game (fig. 4.2) to move the pawn, human players enter first letters of words *up, left, right, down* (*u, l, r, d*). To jump over opponents pawn, combination of simple moves (*uu, ul, ur, dd, dl, dr*) is used. To specify wall placement, first letters of direction are provided (*h, v*) followed by number of the position. For example, *h3, v24* are correctly entered wall positions. Other options available are *undo, load* or *save.*

On the right side of the game board along the options possible, user can see some of the last moves in the HISTORY section. Below the board, there are displayed information such as move number, wall stocks, current distances from the goal and currently moving player.

4.5 PathPlayer

`PathPlayer` is an agent that only follows shortest path to the goal row. I have used this agent as a reference to compare number of won games against `HeuristicPlayer` and `QlearningNetworkPlayer`. User cannot choose to play against this player.

4.6 HeuristicPlayer

`HeuristicPlayer`'s algorithm is following:

- if no walls in stock, follow the shortest path
- if one step left to goal row, move to win
- if standing on the starting line, follow the shortest path
- if there are more than 2 possibilities to block (prolong) opponents path then with 60% probability follow the shortest path
- if there are wall positions blocking opponent's path and not blocking it's own path and it would not cut opponent from goal row, place one such wall
- in all other cases, follow the shortest path

4.7 QLNNPlayer

`QLNNPlayer` evaluates every action in the current state, takes action with maximum value and tries to play it. If move is not possible, it tries to play next best action.

4.8 Training

To speed up the learning proccess, I have created `TrainingStateGenerator` class which provides states to start with for the training. As a game count rises, less trivial states are returned, until only standard initial state is used.

Each time `QLNNPlayer` plays during the training game (inspired by eq. 2.2), desired output vector $\hat{Q}_{new}(s,t)$ is a copy of current output vector $\hat{Q}(s,t)$, where only value representing action performed by `QLNNPlayer` is updated to be eighter one of rewards r_{t+1}, r_{t+2} or maximum of the new q-value estimates $\max_a \hat{Q}(s', t{+}2)$ in the position, where `QLNNPlayer` is on the move again.

In the training, `QLNNPlayer` has probability of playing random move decreasing over time, so that there is a chance to explore and find better moves.

Function `handle_training` running the training along with `TrainingStateGenerator` and `TRAINING_STATES` resides in module `quoridor.ai.training`. It is started through the game menu and it is possible to run training against `PathPlayer` or `HeuristicPlayer`.

Chapter 5

Results

Although I have tested multiple variations of the algorithm, I could not find the right setting for network to learn.

Maybe, input for the network could be better.

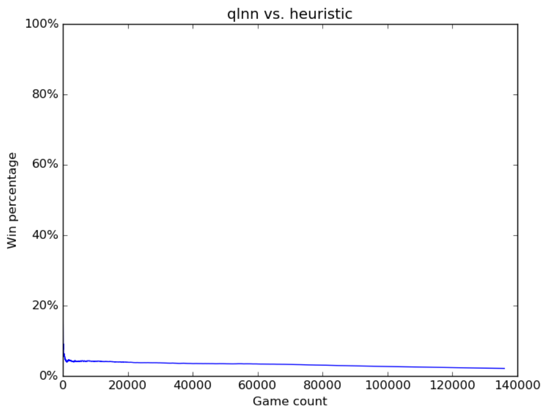


fig. 5.1: ANN(133, 160, 187, 140)

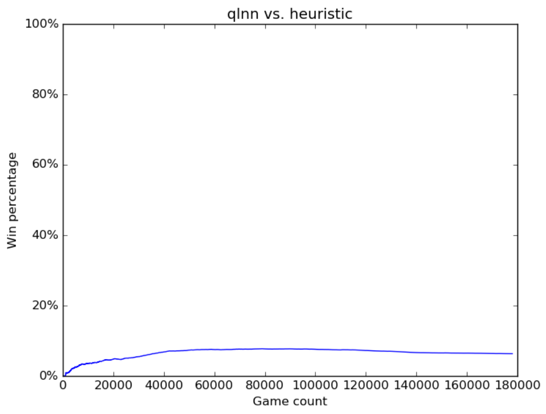


fig. 5.2: ANN(133, 200, 200, 140)

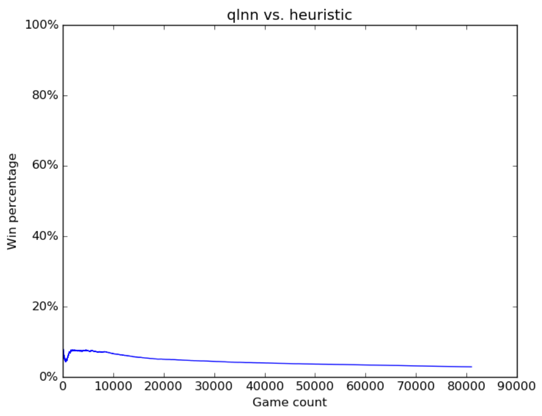


fig. 5.3: ANN(133, 345, 200, 140)

Chapter 6

Conclusion

Although I significantly corrected current estimate of game complexity, still, in terms of creating agent able to play Quoridor utilizing both artifitial neural network with Q-learning method I was was not completely successful, since it did not show signs of notable improvement. It could be necessary to play many more games to start improving or to provide the game state information encoded in the different way.

References

[1] Charles M. Macal and Michael J. North. How to model with agents. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, editors, *Tutorial on Agent-based modeling and simulation part 2*, Argonne, IL 60439, U.S.A., 2006. Winter Simulation Conference.

[2] Michiel van der Ree and Marco Wiering. Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play. April 2013.

[3] P.J.C. Mertens. A quoridor-playing agent, June 2006.

[4] Lisa Glendenning. Mastering quoridor, May 2005.