Quarto! Minimax player
IT3105

Nordmoen, Jørgen H.
Østensen, Trond

October 24, 2012

Abstract

This paper is an introduction to our Quarto! minimax player. In this we will try to explain how we created our implementation, what sort of decisions the player makes the reasoning behind it and our result both against other configurations of our player and against other students minimax implementations.

Contents

List of Figures

1 Introduction

The game of Quarto![1] is a quite simple game regarding its rules, but the hard part comes into play when us humans try to remember all the different attributes of the game. This is where a game playing algorithm comes into play. With an algorithm memory no longer becomes an issue as the algorithm can enumerate, if possible, and search through the whole instance space. In this assignment we were tasked with creating a minimax[2] player with Alpha-beta pruning[3]. The algorithm it self is not the most difficult one to implement, that's not to say that we didn't need to debug our code, but the challenge is coming up with a good set of heuristics to evaluate an intermediate state. We will first introduce our implementation in a birds eye view, we will then explain our chosen heuristics in more detail before we move on to our experience in the tournament. Lastly we will describe our results, both against our own implementation and our tournament results.

2 Implementation

Since we did the last project in Java creating a highly parallel algorithm for roll-out simulation we wanted a different challenge this time around. Because of this we decided that we wanted to create our implementation in Python, but because Python can be quite slow compared to other languages we wanted to implement the time critical Minimax algorithm in C and interface that with our Python code.

Since our implementation had to be supported in both Python and in C we decided quite early on that we needed to represent pieces as simple as possible. After some testing we found that representing each piece as a byte worked very well. Using a byte where each bit represented an attribute gave us a very easy task of sending information between Python and C and made our implementation very fast. There were several advantages representing our pieces as a single byte, not only did it help us interface Python and C, but the representation would not have to change if we needed to play with someone else using this same representation. This is because the method of comparing similarity would not have to change even though we gave different attributes to different bits in the implementation.

Not everything could interface as easily as our pieces though. Since Python is inherently object-oriented our representation of the Quarto! board was also implemented as an object, this did provide us with some headaches which we had to overcome. In the end we managed to get everything working and we got quite good speed out of it which is reflected in our results.

The advantage of doing the project this ways is that we could easily cooperate with other groups playing in Python. And the ease of working in Python is quite different than working in pure C.

As mentioned before we use a byte[4] to represent the individual pieces and to compare equality. All one has to do is "AND" those bytes together to see if

[1] Explanation of rules in video form: www.youtube.com/watch?v=P6dy2eaYmos

[2] Explanation of minimax from Wikipedia: https://en.wikipedia.org/wiki/Minimax

[3] Further explanation on Wikipedia:https://en.wikipedia.org/wiki/Alpha-beta_pruning

[4] In the code we have used integers and unsigned char to represent it in code

all pieces share an attribute. The problem with that approach is that we only capture available attributes not an absent of attributes, e.g. if we have 1100 AND 1010 we only detect that they share the first attribute not the last. To overcome this we took each piece and applied "XOR" against 1111 to produce the compliment. This would mean that the two pieces above also have their compliments compared i.e. 1100 XOR 1111 == 0011 and 1010 XOR 1111 == 0101, 0011 AND 0101 which now will correctly classify the last attribute as shared.

In the following section we might refer to things such as depth, plies and switch. When we are talking about depth or plies we are referring to how many recursion steps that the minimax algorithm should take. When talking about switch we are talking about how many pieces to place with the novice tactics before switching over to the minimax algorithm.

3 Heuristics

The state evaluation function implemented for the minimax algorithm searches through each end-state, looking for instances of board-states that are known to affect the outcome of the match. Our heuristic function is set, arbitrarily, to return a value [-100, 100], where a higher number means a higher valued state.

We have implemented three distinct intermediate state evaluation heuristics and two final state evaluators. The intermediate state evaluators try to assign a value to a state telling us how good or bad that state is if we would end up in that state. The final state evaluators evaluates the final board state telling us if there is a winning player or if the players tied.

3.1 Wins

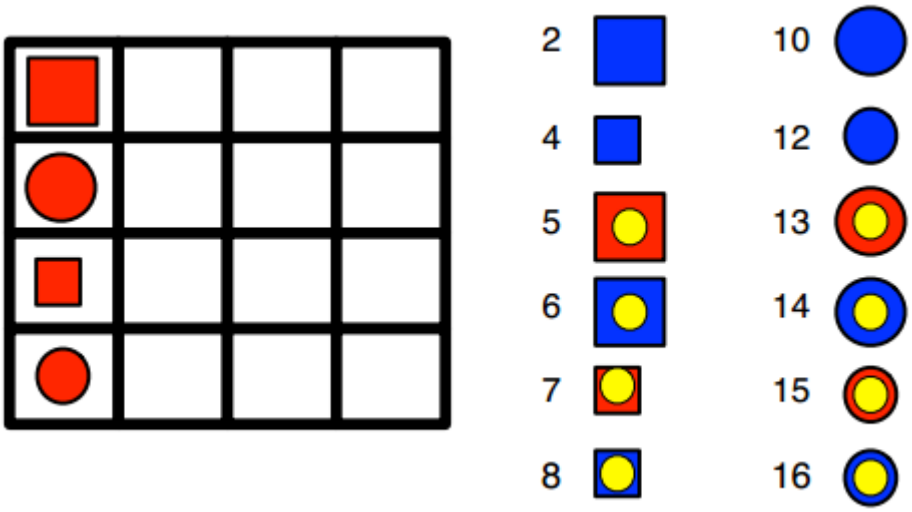

Figure 1: A winning board state in Quarto!

The most obvious board-state to look for is a win(please see figure 1), i.e. four pieces in a line sharing at least one attribute. To do this we compare all the pieces in each row, column and diagonal with each other and if a line is found where all pieces are equal we return a value of 100.

3.2 Ties

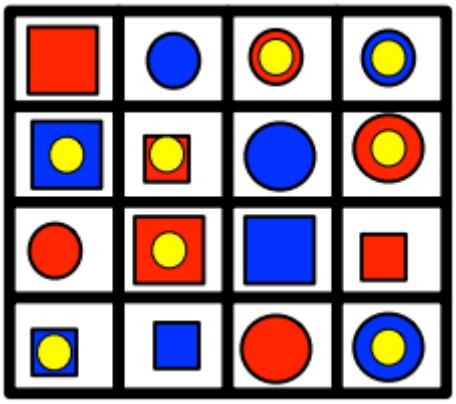

Figure 2: A tied board state in Quarto!

Another nice board-state to look out for is a tie(see figure 2), when all sixteen pieces have been placed on the board, but no win has been achieved. Checking this is merely counting if there are 16 pieces placed on the board, and if there is a win. Since two optimal players will always tie[5], this is evaluated as 0, making sure that the minimax player will always choose a tie over a possible loss.

3.3 Guaranteed losses

When there are two distinct 3-piece lines which have opposite values of at least one attribute(see figure 3), giving any piece to the opponent is a guaranteed loss. When we search for these states, we start of by finding all 3-piece lines on the board, then compare them to see if any of them have opposite values of an attribute. Since a loss is the worst possible outcome of a game of Quarto!, these states are valued at -100 in order to make the minimax player steer away from them.

3.4 3-piece lines

A board containing multiple 3-piece lines makes it difficult to give the opponent a non-winning piece to play. However, depending on whether there is an odd(see figure 4) or even(see figure 5) number of pieces that can be placed in the final slot of a 3-piece line, it is possible to force the opponent into returning a winning piece. With an even number of pieces the opponent can force us into giving him

[5]Proof of optimal play resulting in a tie: http://web.archive.org/web/20041012023358/http://ssel.vub.ac.be/Members/LucGoossens/quarto/quartotext.htm

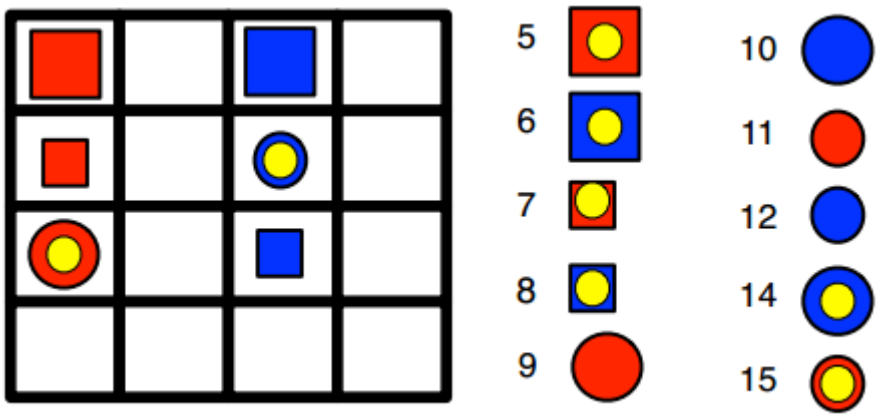

Figure 3: A board state showing a guaranteed loss in Quarto!

a winning piece, and the other way around with and odd number of pieces. Since this is a risky strategy, the 3-piece lines with an even number of winning pieces are weighted heavier than the 3-piece lines with an odd number of winning pieces.

We have had a bit of debate around this last heuristic. Since it can be shown that an optimal player always would tie the last heuristic can't force an opponent into a losing situation, since the player could always finish off the line with a non winning piece before being forced to give away a winning piece. For this reason we were unsure whether or not the heuristics should be included or not. In the end we included it because it should not hinder our performance, but it can, against non optimal players, put us in a situation where we can win.

4 Tournament

When coding for the tournament we did not face many additional challenges, since our internal representation of a Quarto! piece as a byte was consistent with the representation used by the tournament host. And although the mapping of attributes onto bits was chosen different by the players competing in the tournament, the comparison of pieces both internally and externally was never affected by this mapping. The mapping was merely used for displaying board-states visually. What we had to was implementing a class player.py that interfaces with our internal syntax used in $\text{minimax}_{p}layer.pytothesyntaxusedbythetournamenthost.Thiswasnecessaryduetodif$

5 Results

In this section we will describe our results both locally against our self and against two other groups in a tournament setting.

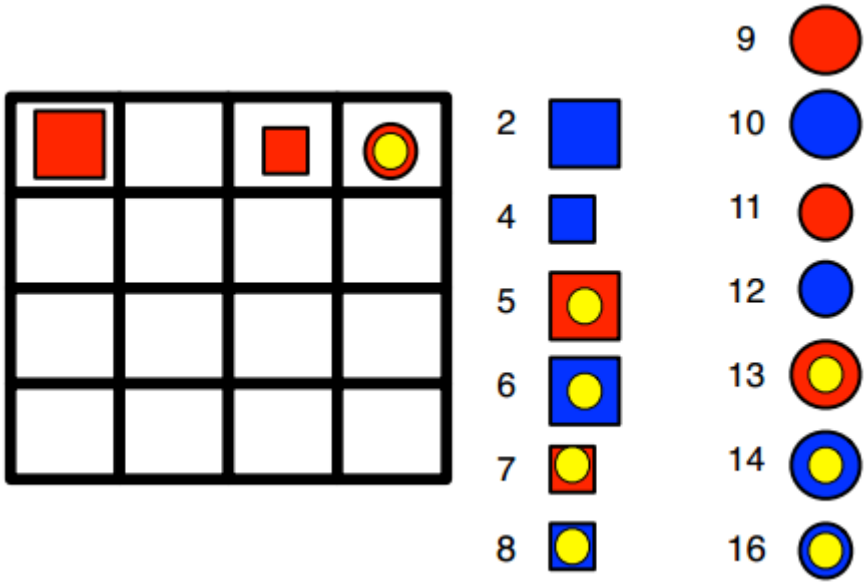

Figure 4: A board state showing a 3-piece line with an odd number of winning pieces

Player type	Wins(%)	Loses(%)	Ties(%)
Random	9 (1%)	488 (97%)	3 (0.6%)
Novice	488 (97%)	9 (1%)	3 (0.6%)

Table 1: Table showing Random compared to Novice player. See listing 1 for command

5.1 Local

One of the first things we wanted to know when we had created our minimax implementation is when we should switch. As the first few positions have very little bearing on the rest of the game, we can play a few moves as a novice before switching to the minimax algorithm. To test when this switch should occur we played our minimax algorithm against it self in a number of scenarios to see when we should switch.

We ran the code in listing 4 and got the results you can see in figure 6. From this we can see that switching on 2, 4 and 6 is really not any different. The small difference we can see is just random variance because of the single run we have tested. When we get down to switching after 8 pieces has been placed there is some difference and we start to see a drop in efficiency.

Below are the results that we got from running our implementation against it self. Each result has a corresponding listing in appendix A where we have added the commands to duplicate our results.

From this results we can see that our Random player is truly living up to its name and losing almost all of the time. Our Novice player can hold its ground against the Random player, but will lose a few times most likely attributed to

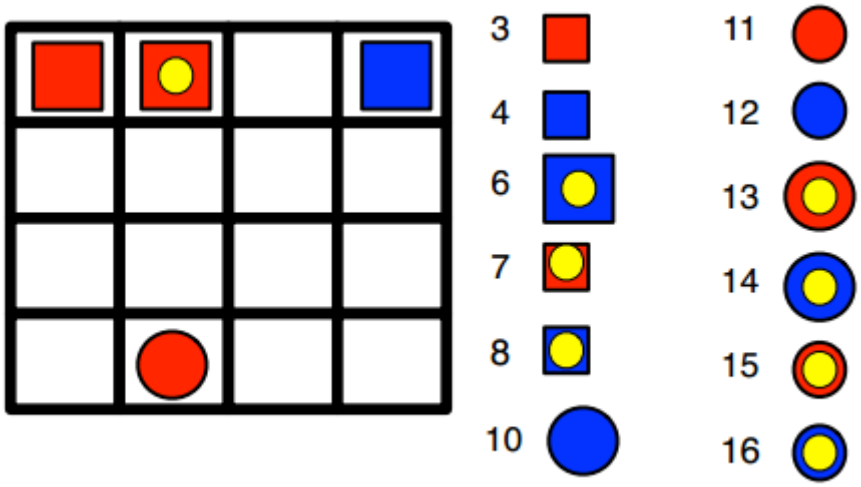

Figure 5: A board state showing a 3-piece line with an even number of winning pieces

Player type	Wins(%)	Loses(%)	Ties(%)
Minimax 3 6	472 (94%)	20 (4%)	8 (1.6%)
Novice	20 (4%)	472 (94%)	8 (1.6%)

Table 2: Table showing Novice compared to Minimax player using depth 3 and switching after 6 pieces placed. See listing 2 for command

it not detecting guaranteed losing situations a head of time. As one can see our Minimax player does quite well and even better when allowed to search further down the tree. It will lose a couple of times against Novice, but this is attributed to the late switching and some states which our heuristics can't detect(Please see section **??** for some extensions which could detect more of these).

5.2 Tournament

The tournament went quite well for us and we did quite good. One of the strengths we had compared to the others were the Python/C implementation which made quite a drastic impact on the time we used in the tournament(as can be seen in table 5). In table 4 we have our raw statistics from the tournament. The tournament was ran for eight(8) hours and each player got a total of 822 games played.

Player type	Wins(%)	Loses(%)	Ties(%)
Minimax 3 6	47 (9%)	207 (41%)	246 (49%)
Minimax 4 6	207 (41%)	47 (9%)	246 (49%)

Table 3: Table showing Minimax 3 6 compared to Minimax 4 6. See listing 3 for command

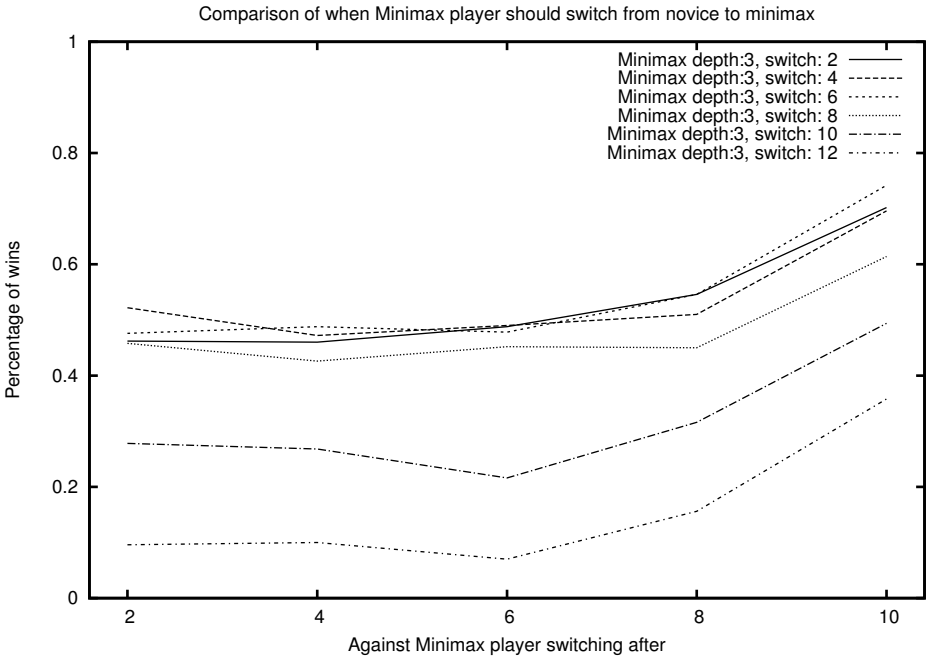

Figure 6: Graph describing our minimax switch results

Player name	Wins(%)	Loses(%)	Ties(%)
Abraham	356 (43%)	238 (29%)	228 (28%)
Mikkel	196 (24%)	444 (54%)	182 (22%)
Trond & Jørgen	366 (45%)	236 (28%)	220 (27%)

Table 4: The tournament result

A Run configuration

We have included our parameters below for the different results that we got in the result section(see section 5). This should enable anyone to clone our git repository and get about the same results out.

Listing 1: Novice compared to random player with 500 games

```
$ python main.py game --player1 random --player2 novice -
    r 500 -s
```

Player name	Time used in seconds
Abraham	7401 (26%)
Mikkel	18250 (63%)
Trond & Jørgen	3352 (11%)

Table 5: Time spent in the tournament calculating

Listing 2: Novice compared to minimax player with 500 games

```
$ python main.py game --player1 minimax 3 6 --player2
    novice -r 500 -s
```

Listing 3: Minimax with a depth of 4 compared to a minimax player with a depth of 3

```
$ python main.py game --player1 minimax 3 6 --player2
    minimax 4 6 -r 500 -s
```

Listing 4: Code to test most minimax depth against each other

```
#!/bin/bash

for i in {2..10..2}
do
        python main.py game --player1 minimax 3 $i --
            player2 minimax 3 12 -r 500 -s >> minimax12.
            txt &
        python main.py game --player1 minimax 3 $i --
            player2 minimax 3 10 -r 500 -s >> minimax10.
            txt
        python main.py game --player1 minimax 3 $i --
            player2 minimax 3 8 -r 500 -s >> minimax8.txt
            &
        python main.py game --player1 minimax 3 $i --
            player2 minimax 3 6 -r 500 -s >> minimax6.txt
        python main.py game --player1 minimax 3 $i --
            player2 minimax 3 4 -r 500 -s >> minimax4.txt
            &
        python main.py game --player1 minimax 3 $i --
            player2 minimax 3 2 -r 500 -s >> minimax2.txt
done
```