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.

[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

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 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.

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.

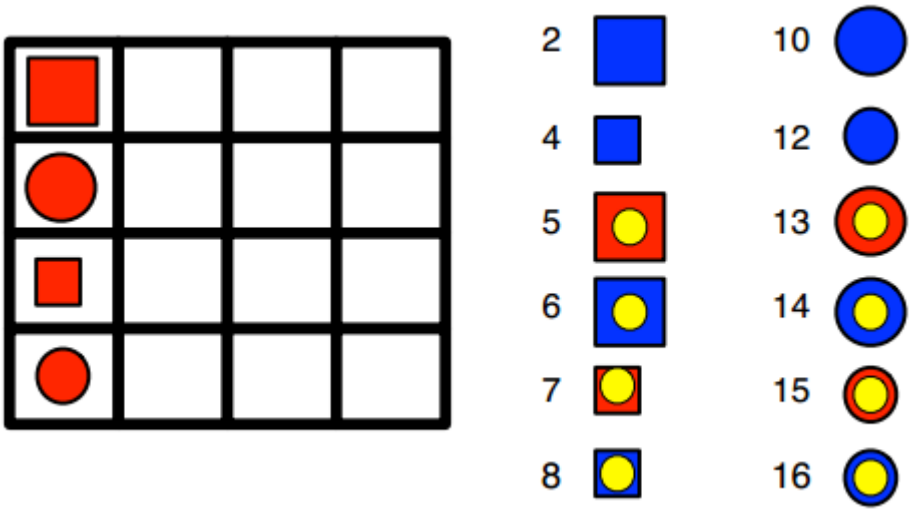

Figure 1: A winning board state in Quarto!

3.1 Wins

The most obvious board-state to look for is a win, 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

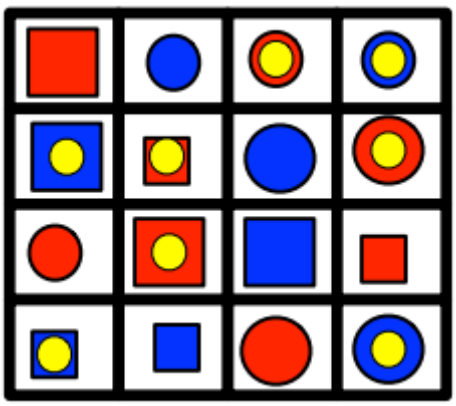

Figure 2: A tied board state in Quarto!

Also a nice board-state to look out for is a tie, when all sixteen pieces have been placed on the board, but no win has been achieved. Checking this is merely

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

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

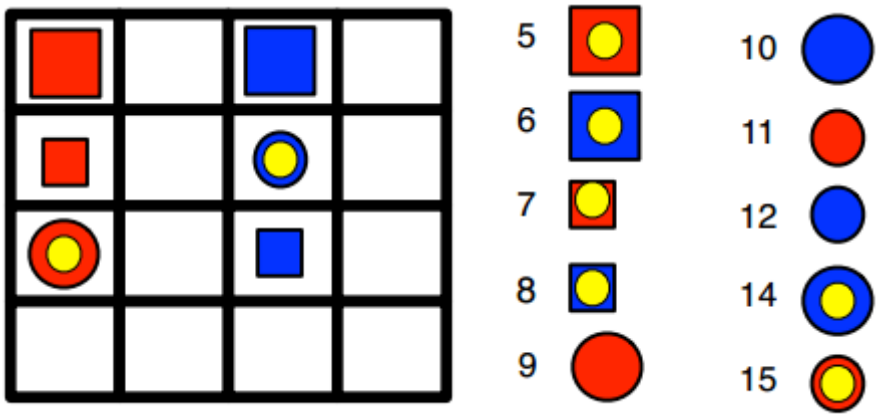

Figure 3: A board state showing a guaranteed loss in Quarto!

When there are two distinct 3-piece lines which have opposite values of at least one attribute, 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 or even number of pieces that can be placed in the final slot of a 3-piece line, is it possible to force the opponent into returning a winning piece. 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.

4 Tournament

5 Results

5.1 Local

5.2 Tournament

[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

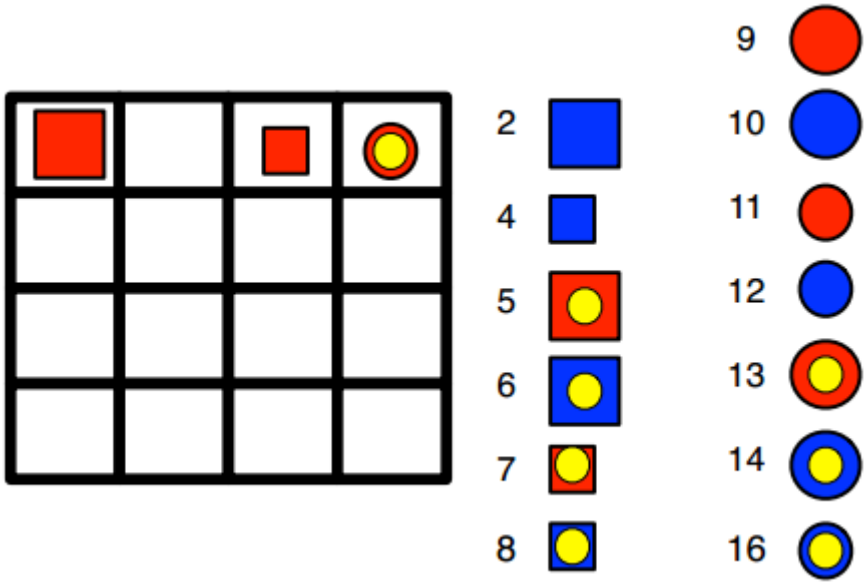

Figure 4: A board state showing a 3-piece line with an odd number of winning pieces

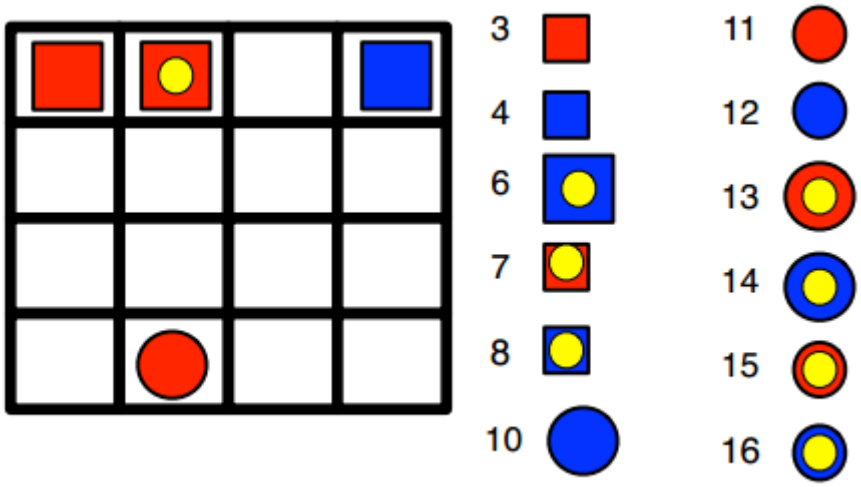

Figure 5: A board state showing a 3-piece line with an even number of winning pieces

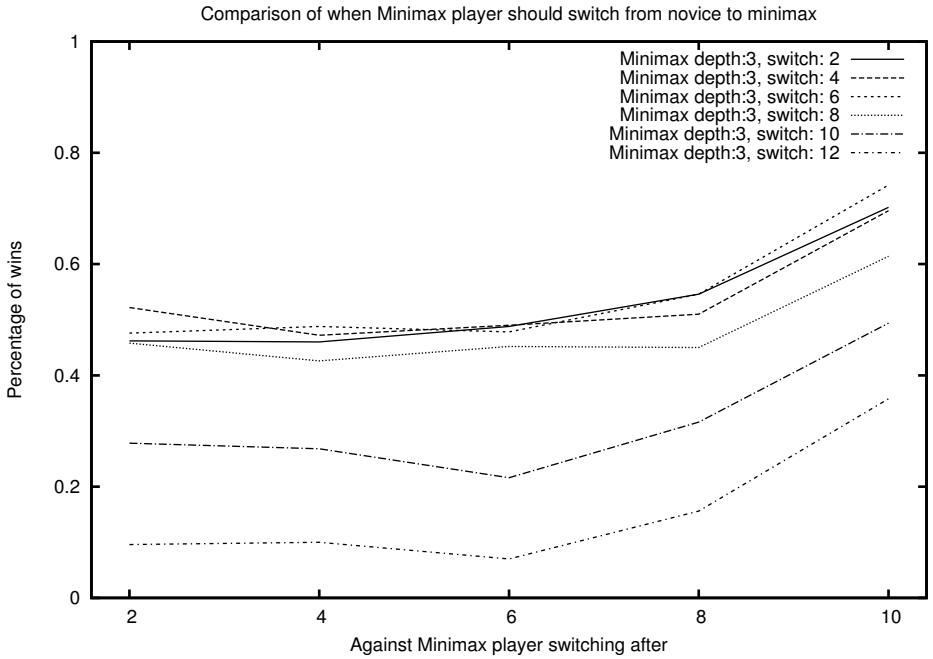

Figure 6: Graph describing our minimax switch results