Report: Assignment 1, Quatrominos

L. Schouten s2660148
D. Macquine s2592991

March 29, 2021

1 Introduction

This report is about an optimal algorithm and a greedy algorithm for the game 'Quatrominos'.

Quatrominos: Game and problem

Quatrominos is a generalized version of the game 'Dominos' and 'Triominos'. Two players play against eachother on a turn base. At the start of the game each player gets an equal amount of tiles. Each tile contains values on the north, east, south and west side of the tile. The game starts with a tile in the middle of the board. The players are allowed to rotate their tiles and may only place their tiles adjacent to other tiles on the board. For a tile to be placed on the board it must have matching values with the adjacent tiles on the board. If a value on one or more sides don't match the values, the tile can't be placed on the board. To win the game a player has to place all their tiles on the board, the player that can't place a tile anymore loses.

For this assignment we had to write an exhaustive search algorithm and a greedy algorithm to determine whether it was possible for the player that was on turn to win the game, assuming both the players would play an optimal strategy. The written algorithm is recursive and uses several other methods to determine if the player on turn can win. The greedy algorithm isn't an exhaustive search algorithm, instead it determines what best move a player can do, a best move leaves the other player with the least amount of possible moves.

Quatrominos: State and action

Quatrominos is played on a flat board. The state of the board can be represented by an $m \times n$ 3D numpy array where two of the dimensions are used for the m rows and n columns and the third dimension is used to hold a tile on a certain position (row, column). The tiles themselves are stored as a 1 dimensional numpy array. A tile consists out of four sides with a value assigned to each side. The order of the sides within the array is: north, east, south, west. Within the 3d numpy array, whenever a spot on the board is still empty, the tile in the 3d numpy array for this position consists of four -1's. An action is defined by placing a tile on a legal position on the board as described in the introduction.

State space diagram: 3×3

Figure 1 shows an example of the state-space on a 3 x 3 board. In this example, the moves of player 0 (current player) are in blue color and the moves of player 1 (other player) are in red. The crosses in the state-space diagram show which player has lost. Initially the board is filled on the given locations with the tiles: [1][0] = (4, 1, 2, 1), [1][1] = (2, 3, 0, 1) and [1][2] = (3, 3, 3,

3). Player 0 starts with the hand {(2, 2, 2, 2), (1, 1, 1, 4)} and player 1 starts with the hand {(5, 3, 1, 5), (1, 4, 1, 4)}. The first player on move is player 0.

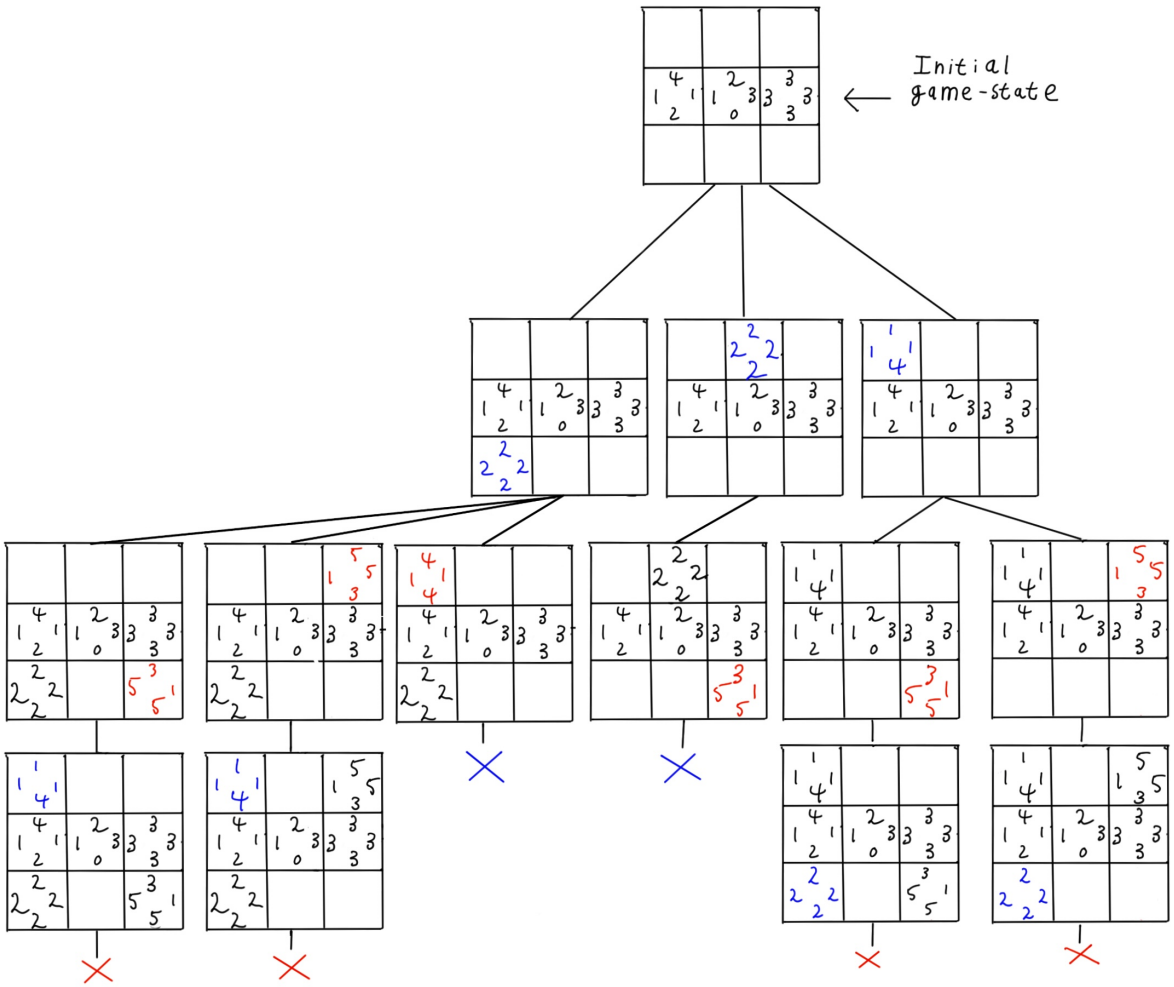

Figure 1: State-space of a 3×3 board

2 Analysis of the optimal Algorithm

It is hard to precisely say how much time it will take for the optimal algorithm to finish. The runtime heavily depends on the size of the board and also on other important factors, like the kind and amount of tiles each player has been assigned. A bigger board means that there might be more possible places to put a tile, more tiles for each player means that there is a bigger chance that one or several of them will fit on one or more places on the board. If one of them will fit, however, also heavily depends on what values are on the their sides. Furthermore, a certain move could open up options for the players. When these moves occur often, the optimal algorithm has to enumerate all of these moves which would increase the time and complexity of the algorithm.

Although it is hard to say how much time it will take for the optimal method to finish, we have tested the running times for this algorithm on a board where both players have 3 tiles, 4 tiles, 5

tiles and 6 tiles:
Using the *get_small_board_adversarial*() board from the test case file the running time for different numbers of tiles for both the players are as follows:

The runtime on a board of 3×3 with each player having 3 tiles is: 0.065 seconds.
The runtime on a board of 3×3 with each player having 4 tiles is: 0.160 seconds.
The runtime on a board of 3×3 with each player having 5 tiles is: 0.423 seconds.
The runtime on a board of 3×3 with each player having 6 tiles is: 0.898 seconds.
The runtime on a board of 3×3 with each player having 18 tiles is: 247.583 seconds.

Using the *get_big_board*() from the test case file the running time for different numbers of tiles for both the players are as follows:
The runtime of a game on a board of 5×5 with each player having 3 tiles is: 0.733 seconds.
The runtime of a game on a board of 5×5 with each player having 4 tiles is: 29.027 seconds.
The runtime of a game on a board of 5×5 with each player having 5 tiles is: 339.086 seconds.
The runtime of a game on a board of 5×5 with each player having 6 tiles is: > 2 hours.

The minimum setup that the optimal method can run within 10 minutes is 5 tiles for each player on the 5×5 board. Giving more tiles, like 6 tiles to each player, will rapidly increase the runtime.

3 Greedy Approach

The greedy approach finds its best move by determining for which move the opponent player has the least amount of possible moves. The greedy approach finds a "best" move quicker than the optimal algorithm but the found move is not always optimal. It is hard to say anything about how much the answer of the greedy approach differs from the optimal approach because the move of the greedy approach is either the same as the optimal move or not. You will never know if the found move is a close to the optimal move or not. An example of when the greedy algorithm does not yield an optimal move is given in the unittest. The adversal unittests "*test_current_player_can_win_greedy_adverserial*()" and
"*test_current_player_can_win_optimal_adverserial*()" are based on the example for the state space diagram 1. In these unittests utilizing the greedy move that leaves the other player with the least amount of moves results in a lose for the current player. But utilizing the optimal move would lead to a win for the current player. In the unittest the greedy move for the current player would be to place (2, 2, 2, 2) on [0][1] on the board because this leaves the other player with only one possible move. When the other player does this move we lose. The optimal move is placing (1, 1, 4, 1) on [0][0]. This move will be determined with the current_player_can_win() function and the state-space diagram shows that this is the only way for the current player to win.

4 Summary

This report is about an optimal algorithm and a greedy algorithm for the game 'Quatrominos'. The optimal algorithm is an exhaustive search algorithm that uses a recursive function to determine if the current player can win when playing optimal. The greedy algorithm determines a best move by checking what move leaves the other player with the least amount of options to play. The optimal works as expected for an exhaustive search algorithm, it finds the "perfect"

move when there is one and the running time increases rapidly with the number of possibility's the optimal move has to enumerate. The greedy approach is faster than the optimal algorithm and makes an educated guess. However, the greedy method isn't always optimal.