The Best Move is the Least Bad: an AI for 2048
MATH 4500 Final Report
Clemson University

Duncan Nicholson

May 2, 2019

Contents

1 Introduction **1**

2 Methods **1**

3 Results **2**

A Code implementation **4**

1 Introduction

The game of *2048* is a 4x4 sliding block puzzle designed by Italian web developer Gabriele Cirulli .The 19-year-old Cirulli reportedly wrote the game using *javascript* and *CSS* in a single weekend. Within a short time, *2048* had gained over 5 million players on IOS and Android and gained a fiendishly addictive reputation. You can try the game for yourself at `https://play2048.co/`

Due to the frustration and wasted time the game caused the author back in 2014 when it was released, it was figured that the game could serve as a gentle introduction to one or more simple artificial intelligence (AI) algorithms. The basic plan, in experimental mathematical fashion, was to throw computer codes at the game repeatedly to see what came of it, and also to determine wether the game could be beaten repeatably. If this plan failed, at the very least the author would have arisen with a substantially increased knowledge of coding in Python, the language of choice for MATH 4500.

2 Methods

Since the author is first by trade an engineering student (and mathematician second) they had little desire to implement any algorithms that required too

much analysis of the game itself. Initial research showed that the best performing AI's employed *heuristics* that were hard-coded into the AI by the author - in otherwords, observations related to winning gameplay. Some of these heuristics were explained in the proposal for this project (notably: number of free tiles on the board as well as monotonicity. The best performing algorithm found, *expectimax*, makes use of these in heuristic scoring functions).

The game was first implemented using manual input from the keyboard, resulting in the move() function, which can be found in the code contained in the appendix of this document.

In an attempt to implement an AI from which the author would be able to take more skills away from, an algorithm that contained no hard-coded intelligence or human understanding of the game was implemented. Loose guidance was found from an algorithm called Pure Monte Carlo game Search.

The implementation for *2048* is summarized as follows. To determine the best move for any given game state, the AI first copies the board into memory. Then, for all possible starting moves, "runs" number of games are played to completion using random moves. The score of these randomly played games (the sum of the tiles) are averaged for each starting move. The best move is chosen as the starting move with the highest average score.

As the number of games played in memory ("runs") approaches infinity, the AI should approach perfect gameplay.

3 Results

The python implementation was found to achieve a 2048 tile (winning the game) repeatably using around 100 in memory games per move. Figure 1 shows typical results of the algorithm using 3, 10, and 100 runs, respectively.

Sage/the cocalc server was not used for the final calculations because the algorithm (or my code implementation) is extremely slow. The winning run with runs=100 took approximately 35 minutes running on a single core pegged at 100%. Higher tiles are definitely achievable, but multithreading is recommended for time improvement.

The implementation was considered a success because the AI beat the game and a huge amount of practice in python/sage was achieved.

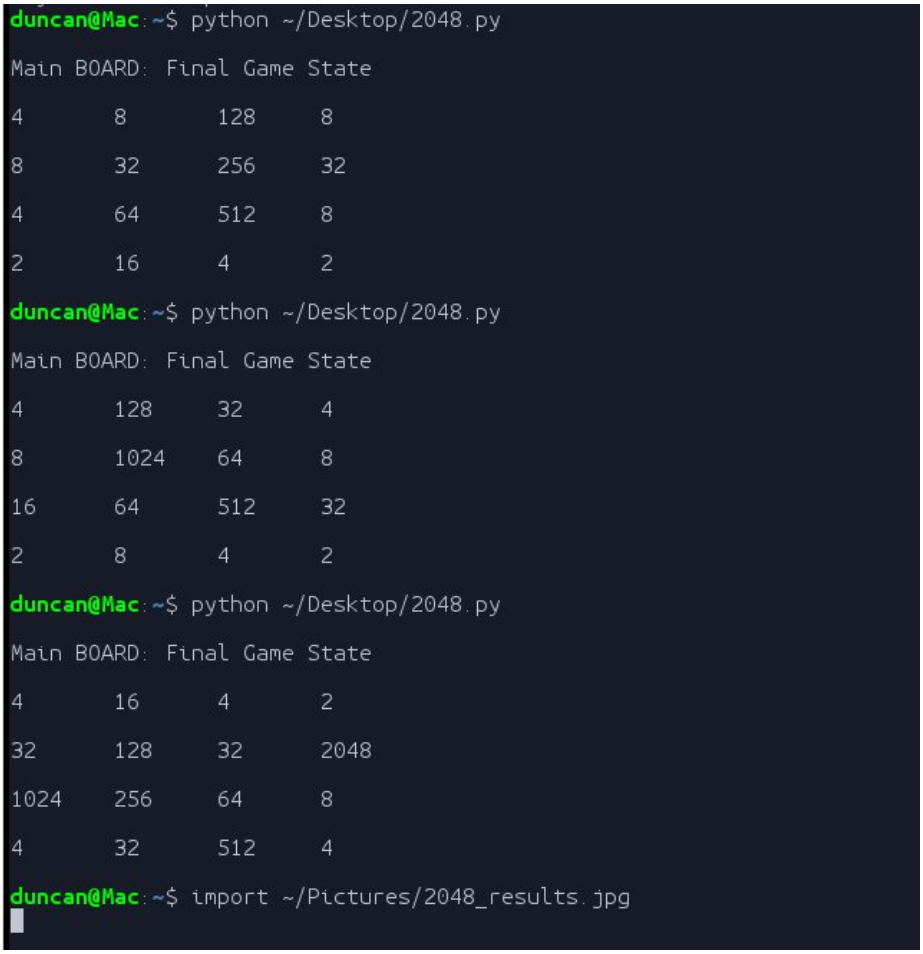

Figure 1: Game winning results of the AI for runs=3,10,100

A Code implementation

```
0024bcb1-2734-4740-a62f-ddcd30fa5e3cr
import random
import math
import operator
import numpy as np
import copy

#random.seed(12345)
sage_server.MAX_OUTPUT=1e6

def spawn(N):
    #check for open tiles
    listforn = []
    listform = []
    zeros = 0
    for n in range(0,4):
        for m in range(0,4):
            if N[n][m]==0:
                zeros=zeros+1
                listforn.append(n)
                listform.append(m)
    if zeros > 0:
        if zeros >=1:
            randomindex = listforn.index(random.choice(listforn))
            Pspawn = random.choice(srange(5))/5
            if Pspawn >= .8:
                N[listforn[randomindex]][listform[randomindex]]=4
            else:
                N[listforn[randomindex]][listform[randomindex]]=2
    return N;

def print_board(N):
    print N[0][0],"\t",N[0][1],"\t",N[0][2],"\t",N[0][3],"\n"
    print N[1][0],"\t",N[1][1],"\t",N[1][2],"\t",N[1][3],"\n"
    print N[2][0],"\t",N[2][1],"\t",N[2][2],"\t",N[2][3],"\n"
    print N[3][0],"\t",N[3][1],"\t",N[3][2],"\t",N[3][3],"\n"
    return

def move(B,input):
    if input == "u":
        n=0
        # loop over columns
        for m in range(0,4):
            # if any on a column are not zero:
```

```
            if B[n][m]!=0 or B[n+1][m]!=0 or B[n+2][m]!=0 or B[n+3][m]!=0:
                # if the first element in the column is zero
                if B[n][m]==0:
                    # while the top tile is zero
                    while B[n][m]==0:
                        #assign tile below's value to top tile in column
                        #Get rid of these hardcoded indices
                        B[n][m]=B[n+1][m]
                        B[n+1][m]=B[n+2][m]
                        B[n+2][m] = B[n+3][m]
                        B[n+3][m]=0
                if B[n+1][m]==0 and (B[n+2][m]!=0 or B[n+3][m]!=0):
                    while B[n+1][m]==0:
                        B[n+1][m]=B[n+2][m]
                        B[n+2][m]=B[n+3][m]
                        B[n+3][m]=0
                if B[n+2][m]==0 and (B[n+3][m]!=0):
                    while B[n+2][m]==0:
                        B[n+2][m]=B[n+3][m]
                        B[n+3][m]=0
        n=0
        for m in range(0,4):
            if B[n][m]==B[n+1][m]:
                B[n][m]=B[n][m]+B[n+1][m]
                B[n+1][m]=B[n+2][m]
                B[n+2][m]=B[n+3][m]
                B[n+3][m]=0
            if B[n+1][m]==B[n+2][m]:
                B[n+1][m]=B[n+1][m]+B[n+2][m]
                B[n+2][m]=B[n+3][m]
                B[n+3][m]=0
            if B[n+2][m]==B[n+3][m]:
                B[n+2][m]=B[n+2][m]+B[n+3][m]
                B[n+3][m]=0

    #user inputs down
    elif input == "d":
        # in this loop we collapse the matrix in the specified direction
        n=0
        #loop over columns
        for m in range(0,4):
            # if any tiles in that column are not zero
            if B[n][m]!=0 or B[n+1][m]!=0 or B[n+2][m]!=0 or B[n+3][m]!=0:
                # if bottom tile in column is zero
                if B[n+3][m]==0:
                    # collapse column completely
```

```
                while B[n+3][m]==0:
                    B[n+3][m]=B[n+2][m]
                    B[n+2][m]=B[n+1][m]
                    B[n+1][m]=B[n][m]
                    B[n][m]=0
            # if second from bottom tile is zero and nonzero above
            if B[n+2][m]==0 and (B[n+1][m]!=0 or B[n][m]!=0):
                # collapse column to second from bottom tile
                while B[n+2][m]==0:
                    B[n+2][m]=B[n+1][m]
                    B[n+1][m]=B[n][m]
                    B[n][m]=0
            # if top tile in column can slide down one space
            if B[n+1][m]==0 and B[n][m]!=0:
                # move top tile down one
                while B[n+1][m]==0:
                    B[n+1][m]=B[n][m]
                    B[n][m]=0
    n=0
    # This loop merges identical tiles in the specified direction
    # loop over columns again
    for m in range(0,4):
        if B[n+3][m]==B[n+2][m]:
            B[n+3][m]=B[n+3][m] + B[n+2][m]
            B[n+2][m]=B[n+1][m]
            B[n+1][m]=B[n][m]
            B[n][m]=0
        if B[n+2][m]==B[n+1][m]:
            B[n+2][m]=B[n+2][m]+B[n+1][m]
            B[n+1][m]=B[n][m]
            B[n][m]=0
        if B[n+1][m]==B[n][m]:
            B[n+1][m]=B[n+1][m]+B[n][m]
            B[n][m]=0

elif input == "l":
    m=0
    for n in range(0,4):
        if B[n][m]!=0 or B[n][m+1]!=0 or B[n][m+2]!=0 or B[n][m+3]!=0:
            if B[n][m]==0:
                while B[n][m]==0:
                    B[n][m]=B[n][m+1]
                    B[n][m+1]=B[n][m+2]
                    B[n][m+2] = B[n][m+3]
                    B[n][m+3]=0
        if B[n][m+1]==0 and (B[n][m+2]!=0 or B[n][m+3]!=0):
```

```
                while B[n][m+1]==0:
                    B[n][m+1]=B[n][m+2]
                    B[n][m+2]=B[n][m+3]
                    B[n][m+3]=0
            if B[n][m+2]==0 and (B[n][m+3]!=0):
                while B[n][m+2]==0:
                    B[n][m+2]=B[n][m+3]
                    B[n][m+3]=0

        m=0
        for n in range(0,4):
            if B[n][m]==B[n][m+1]:
                B[n][m]=B[n][m]+B[n][m+1]
                B[n][m+1]=B[n][m+2]
                B[n][m+2]=B[n][m+3]
                B[n][m+3]=0
            if B[n][m+1]==B[n][m+2]:
                B[n][m+1]=B[n][m+1]+B[n][m+2]
                B[n][m+2]=B[n][m+3]
                B[n][m+3]=0
            if B[n][m+2]==B[n][m+3]:
                B[n][m+2]=B[n][m+2]+B[n][m+3]
                B[n][m+3]=0

    elif input == "r":
        m=0
        #loop over rows
        for n in range(0,4):
            if B[n][m]!=0 or B[n][m+1]!=0 or B[n][m+2]!=0 or B[n][m+3]!=0:
                if B[n][m+3]==0:
                    while B[n][m+3]==0:
                        B[n][m+3]=B[n][m+2]
                        B[n][m+2]=B[n][m+1]
                        B[n][m+1]=B[n][m]
                        B[n][m]=0
            if B[n][m+2]==0 and (B[n][m+1]!=0 or B[n][m]!=0):
                while B[n][m+2]==0:
                    B[n][m+2]=B[n][m+1]
                    B[n][m+1]=B[n][m]
                    B[n][m]=0
            if B[n][m+1]==0 and B[n][m]!=0:
                while B[n][m+1]==0:
                    B[n][m+1]=B[n][m]
                    B[n][m]=0
        m=0
        for n in range(0,4):
```

```
            if B[n][m+3]==B[n][m+2]:
                B[n][m+3]=B[n][m+3] + B[n][m+2]
                B[n][m+2]=B[n][m+1]
                B[n][m+1]=B[n][m]
                B[n][m]=0
            if B[n][m+2]==B[n][m+1]:
                B[n][m+2]=B[n][m+2]+B[n][m+1]
                B[n][m+1]=B[n][m]
                B[n][m]=0
            if B[n][m+1]==B[n][m]:
                B[n][m+1]=B[n][m+1]+B[n][m]
                B[n][m]=0
    return B

def check_possible(N, input, b):
    # Move is possible if, for any row, there are zeros in front of a number, or there are t
        # return true if, for any check_vec
                    # tile after zero
                    # adjacent identical tiles

    #extract vectors to check
    check_vecs=[[] for _ in srange(b)];
    if input == "u":
        # for each check vec needed
        for j in range(0,b):
            #extract the check vec
            for i in srange(0,b):
                check_vecs[j].append(N[i][j])

    elif input == "d":
        #same as up
        for j in range(0,b):
            for i in srange(0,b):
                check_vecs[j].append(N[i][j])
        #but reverse them
        i=0
        for i in range(0,b):
            list.reverse(check_vecs[i])

    elif input == "l":
        for j in range(0,b):
            for i in range(0,b):
                check_vecs[j].append(N[j][i])

    elif input=="r":
        for j in range(0,b):
```

```
        for i in range(0,b):
            check_vecs[j].append(N[j][i])
    i=0
    for i in range(0,b):
        list.reverse(check_vecs[i])
#print("check_vecs = ");print(check_vecs);

#check the vectors
for vec in check_vecs:
    #print("checking vector:");print(vec)

    #if no tile()s) in vec, all zeros, no move possible, move on to next check vec
    if np.any(vec)==False:
        continue

    # find the first zero
    zero_spot=[]
    try:
        zero_spot = vec.index(0)
    except ValueError:
        #print('no zeros in vec')
        zero_spot=[]

        i=0
        for i in range(0,b-1):
            if vec[i]!=0 and vec[i+1]!=0:
                if vec[i]==vec[i+1]:
                    #we have a pair, move is possible
                    #print("mergable pair found")
                    return True

    #if there is a zero, check for tile after zero
    if zero_spot !=[]:
        i=0
        for i in range(zero_spot+1,b):
            #if hit a number, move is possible
            if vec[i]!=0:
                return True

    #always check for mergers
    i=0
    for i in range(0,b-1):
        if vec[i]!=0 and vec[i+1]!=0:
            if vec[i]==vec[i+1]:
                #we have a pair, move is possible
                #print("mergable pair found")
```

```
                    return True
    # if you made it out of the check_vecs[vec] for loop, selectedmove is not possible
    return False

def game_over(N,b):
    #print("## checking if game is over ###");
    if check_possible(N,"u",b)==False and check_possible(N,"d",b)==False and check_possible(
        #print("game over! result:")
        #print_board(N)
        return True
    else:
        return False

def score_board(N):
    score=0.0
    for n in range(0,4):
        for m in range(0,4):
            if N[n][m]!=0:
                c = N[n][m]
                score=score + c
    return score

def main(runs):
    #print("score format: [up down left right]");print("")

    # Initialize game board #size bxb
    b=4
    A = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]];

    #spawn first tile
    A=spawn(A)

    while game_over(A,b)==False:

        starting_move = []
        scores=[]

        #print("######## MAIN BOARD ##########");
        #print_board(A)
        #print("################################");
        #print("")

        for starting_move in ["u","d","l","r"]:

            # make a ***copy*** of board. A_=A is only references A and calling A_=move(A_)
            A_=copy.deepcopy(A)
```

```
            if check_possible(A_,starting_move,b)==True:
                #print("starting_move");print(starting_move)
                A_=move(A_,starting_move)
                A_=spawn(A_)

                random_scores=[]
                m=0
                #play runs number of games using random moves and average the final scores
                for m in srange(0,runs):
                    A__=copy.deepcopy(A_)
                    while game_over(A__,b)==False:
                        random_move=random.choice(["u","d","l","r"])
                        if check_possible(A__,random_move,b)==True:
                            A__=move(A__,random_move)
                            A__=spawn(A__)
                    random_scores.append(score_board(A__))

                scores.append(sum(random_scores)/runs)
            else:
                #print('starting move not possible')
                scores.append(0)
                continue

        # print scores for simulated games
        #print("scores:");print(scores);print("")

        # find move which netted the highest score
        index, value = max(enumerate(scores), key=operator.itemgetter(1))
        #print("index");print(index);print("")

        # write result of tree search to actual game board
        if index ==0:
            auto_input="u"
        elif index ==1:
            auto_input="d"
        elif index ==2:
            auto_input ="l"
        elif index ==3:
            auto_input ="r"
        #print("auto generated input:");print(auto_input);print("")
        # make real move
        A=move(A,auto_input)
        A=spawn(A)

    print("")
```

```
    print("Main BOARD: Final Game State")
    print("")
    print_board(A)

main(runs = 100)

491794df-9685-4233-aa49-b42808f90294
1930ecbb-e5be-4e2a-b7ad-742e25a9076d

937836b7-dff6-4969-9035-00d4a67990e5
1961d8b0-4109-497b-a817-e8e7b84a9984
```