Towards a Hybrid Neural and Evolutionary Heuristic Approach for Playing Tile-matching Puzzle Games

Jose M. Font
Faculty of Technology and Society
Malmö University
jose.font@mah.se

Daniel Manrique
ETS Ingenieros Informáticos
Universidad Politécnica de Madrid
dmanrique@fi.upm.es

Sergio Larrodera
ETS Ingenieros Informáticos
Universidad Politécnica de Madrid
s.larrodera@alumnos.upm.es

Pablo Ramos Criado
ETS Ingenieros Informáticos
Universidad Politécnica de Madrid
pablo.ramos.criado@alumnos.upm.es

***Abstract*— In this paper we explore the hybrid application of evolutionary computation and artificial neural networks in the development of intelligent systems able to solve the problem of approximating the optimal strategy in a tile-matching puzzle game. Three intelligent systems are proposed: an evolutionary heuristic technique, artificial neural networks, and a hybrid approach that combines both. Results show that the hybrid approach, which combines the advantages of the two previous solutions, performs better at both, the number of completed lines and the average piece placement time. These results aim to serve as the basis for a later comparative study against state-of-the-art techniques in the topic.**

***Index Terms*— Evolutionary algorithms, Neural Networks, AI agents, Tetris**

I. INTRODUCTION

Tetris popularized tile-matching puzzle games in the 1980's [1]. In the original game, a single player must place square-based pieces that sequentially fall at an increasing speed from the upper row in a squared 20 by 10 tiles board. Only rotations and horizontal shifts can be applied to the pieces before they reach the bottom of the board. Rows are removed right after the player manages to fill them in by correctly placing the falling pieces, being the player rewarded after completing a row. The game ends when the pile of pieces is higher than the board.

Figure 1 displays (in every possible rotation) the seven existing pieces in the game (tetrominoes, the subset of the polyominoes [2] with size four). Each of the pieces is identified by a letter that resembles its shape [3].

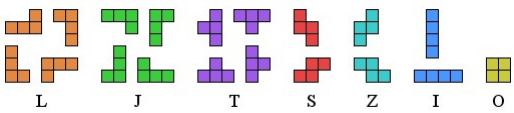

Fig. 1. Existing pieces in every possible rotation, and their corresponding similar shaped letters.

The sequence of falling pieces is selected by the game at random, which classifies Tetris as a game against nature [4]. Since there is no theoretical limit to the length of the game, any given sequence will eventually appear. This means that the player will always lose the game, regardless of the strategy followed [5].

Tetris has captured the interests of computer gamers and Artificial Intelligence (AI) researchers [6][5] both as a testbed for novel techniques [7][8][9][10], as well as an application for building intelligent agents for finding optimal winning strategies in the game [11][12]. The general approach to developing controllers for Tetris is a minimax based search and evaluation [13] adapted for one-player games. It is assumed that the game always chooses its next move (the next falling piece to place) at random. Depending on the game implementation, the search process relies on a *one-piece* or *two-piece* evaluation function. The first case assumes that the player only knows the current state of the board as well as the current piece to place. The second one adds the information about the next piece to be played.

A deep overview of the existing computational controllers developed for Tetris can be found in [12]. The most notable contributions include a hand-coded evaluation function [14] that includes a linear combination of five game board features whose weights are manually tuned by trial and error. Evolutionary techniques have been also presented for optimizing the weights in the evaluation functions according to the average number of completed lines of several playouts [15][11]. Other approaches apply reinforcement learning techniques [16][10] or ant-colony optimization [17] instead. Neural Networks, widely used for implementing controllers for a varied set of games [18][19][20][21][22], have been proposed combined with reinforcement learning for creating a Tetris agent [3].

The following sections describe and present a comparative study on three different approaches for calculating optimal strategies for AI agents that play Tetris: an evolutionary heuristic approach [23], artificial neural networks (ANNs)

[24], and a hybrid agent that combines both. The three of them operate under the original game settings using a one-piece evaluation method.

II. THE EVOLUTIONARY HEURISTIC, NEURAL AND HYBRID APPROACHES

A. The Evolutionary Heuristic

The controller uses a one-step lookahead maximization heuristic for calculating the optimal placement for a given piece. Given the current game state and the next piece to place, the controller calculates all the possible (legal) placements for that piece. The heuristic then approximates the quality of each of these simulated states, returning the placement that leads to the next best state. The proposed controller operates on eleven representative features extracted from the game state:

- c_1 The square sum of heights: computed as the square root of the sum of the squares of the column heights. This allows the detection of placements that complete lines.
- c_2 Height variation: the sum of the height variation from each column to the subsequent one. This favors flat board states.
- c_3 The square height variation: the square root of the sum of the height variations. Favors flat board states with a greater penalty to uneven columns.
- c_4 Holes: number of holes inside the columns.
- c_5 Weighted holes: each hole's penalty is calculated as $p = \frac{a+b}{d^2}$, where a is the row the hole is located in, b is the number of block tiles above the hole, and d is the distance to the closest block tile above it. This computation increases the penalty for holes with large piles of block tiles above them. Feature value is calculated as the sum of all penalties.
- c_6 Upstairs: equals 0 if the height variation between any pair of subsequent columns is 1, otherwise, $c_6 = 1$.
- c_7 Downstairs: equals 0 if the height variation between any pair of subsequent columns is -1, otherwise, $c_7 = 1$.
- c_8 Plateaus: equals 0 if the height variation between any pair of subsequent columns is 0, otherwise, $c_8 = 1$.
- c_9 2-2 pits: square of the number of occurrences where three consecutive columns have height variations of, from left to right, -2 and 2.
- c_{10} 2-3 pits: square of the number of occurrences where three consecutive columns have height variations of, from left to right, -2 and 3, or -3 and 2.
- c_{11} 3-3 pits: square of the number of occurrences where three consecutive columns have height variations of, from left to right, -3 and 3.

These features are linearly combined in order to evaluate any given game board S as:

$$eval(S) = \lambda_1 \cdot c_1(S) + \lambda_2 \cdot c_2(S) + ... + \lambda_{11} \cdot c_{11}(S), \quad (1)$$

where $\lambda_i \in [0, 1]$, $\forall i = 1, ..., 11$, and $\Sigma_{i=1}^{11} \lambda_i = 1$ are the weights for the features $c_1, ..., c_{11}$ in the state S.

A real-valued GA is applied to obtain an optimal weights set that maximizes the controller performance. The GA individuals are vectors containing heuristic weights sets. Individuals' fitness is obtained from the controller performance using individuals' heuristic weights set. The GA uses a population of 24 individuals, initially generated at random by sampling values in the range $[0, 1]$. Individuals are normalized so that all weights always sum 1. The morphological crossover operator [23], tournament parent selection, and elitist replacement have been employed.

The fitness calculation is a non-deterministic process, since the same individual can perform very differently in subsequent playouts. For this reason, every individual is evaluated once at every generation. The latest fitness score is cumulated with the previous one by following an aging fitness function. For a given individual I_i and generation t, we define the aging fitness function $g(I_{i,t})$ as follows:

$$g(I_{i,t}) = \frac{age_{i,t} \cdot g(I_{i,t-1}) + score_{i,t}}{age_{i,t} + 1}, \quad (2)$$

where $g(I_{i,t-1})$ is the fitness in the previous generation, $score_{i,t}$ is the number of lines scored in the current playout, and $age_{i,t}$ is the number of playouts performed by the individual I_i so far; that is, the number of times this individual has been evaluated. This aging function sequentially increase the fitness accuracy evaluation after evaluation. Consequently, those individuals that underperform are easily revealed and discarded. The genetic algorithm stops when the population average fitness do not improve in 50 generations.

Table I displays the average results from this approach out of 1000 executions run. Note that, due to the nature of the problem, the average number of completed lines is highly variable. Such a large sample size explains why the size of the the confidence interval looks small when compared to the standard deviation. Some of the optimal individuals produced reached up to $287,418$ lines, but the confidence interval shows that these results were quite sporadic.

TABLE I
AVERAGE RESULTS FROM THE EVOLUTIONARY HEURISTIC

Average piece placement time (ms)	0.084066
Lines	
Average	36037
Standard deviation	36667
95% confidence interval	[33764, 38309]
Min	39
Max	287418

Additional experiments have been performed with a two-step lookahead maximization algorithm (named simulation T=1). This means that the controller expands the simulated end states tree one level deeper. After obtaining every possible end state given the current piece, the seven possible pieces that may appear (Figure 1) are considered to simulate all the reachable states in an additional turn. Though this extensions raises the chance of getting better results in terms of the average completed lines, the average piece placement time raises to $11.46ms$, which leads to unacceptably long

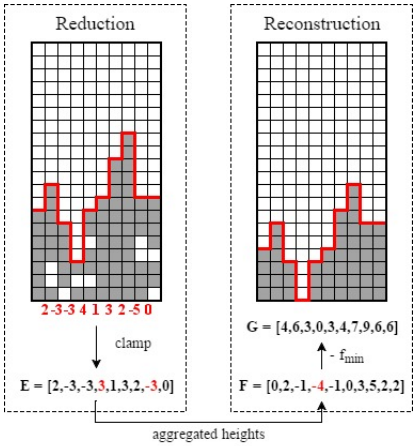

Fig. 2. Reduction and reconstruction process of the game board.

computing times for a game controller that is expected to perform in real game time.

B. The Neural Network Set

Though simulation T=1 would considerably improve the heuristic results, it cannot operate in real time. For this reason, a set composed by seven neural networks, one per piece type (Figure 1), was trained in order to make each network learn where to place (column and rotation) a specific piece. Each network has as many output neurons (n) as possible placements for its related piece. A network outputs a probability vector $P = [p_1, ..., p_n]$, $\sum_{i=1}^{n} p_i = 1$. This vector indicates the probability p_i for each placement i (column and rotation) to be the most suitable one. When the next falling piece to place appears at the top of the board, its corresponding network receives and processes the board state, so that the highest output value (probability) is selected as the optimal placement. Hidden and output neurons use *rectifier linear* and *softmax* activation functions, respectively. The networks are trained by backpropagation.

The 200 tiles original board requires 200 input neurons per network. To reduce the complexity, the input is encoded as a real-valued vector with dimension 9, in which each value (from -3 to +3) represents the height difference from one column to the next one in a game state. The height is measured in tiles, being -3 and 3 the highest negative and positive distances respectively. Though this causes loss of information, it can be assumed as a trade-off for significantly reducing the input complexity from 2^{200}, resulting from 200 binary tiles (either they contain a block tile or they are empty), to 7^9.

Figure 2 shows the reduction and reconstruction process that is applied to game boards. The vector $E = [e_0, ..., e_8]$ encodes the board as a sequence of height variations of subsequent columns. Numbers in red for this vector in the example shown in Figure 2 indicate that the height variation in the original board was greater than 3 or less than -3. During reconstruction, this vector is translated to $F = [f_0, ..., f_9]$: the cumulated heights vector in which $f_0 = 0$ and each subsequent position $f_i = f_{i-1} + e_{i-1}$. Finally, the target height vector $G = [g_0, ..., g_9]$ is reconstructed as $g_i = f_i - f_{min}$, where f_{min} is the lowest value in F (red-marked in Figure 2). Since only the height of the columns are encoded, all the middle holes (empty tiles with pieces above) in the original board are lost in the reconstruction process. This scenario is also illustrated in the example given in Figure 2.

The evolutionary heuristic with simulation T=1 was run off-line 100 times in order to generate the training set for the neural networks. The average number of completed lines achieved was 41655, with a 95% confidence interval between $[33855, 49456]$, reaching 182078 completed lines in the best case. The generated datasets contained 100000 instances per piece type.

TABLE II
AVERAGE RESULTS, OUT OF 1000 PLAYOUTS, FROM THE BEST NEURAL NETWORK SET TRAINED WITH SIMULATION T=1

Average piece placement time (ms)	0.166092
Lines	
Average	9429
Standard deviation	8994
95% confidence interval	[8872, 9987]
Min	52
Max	50068

A set of pre-runs using cross-validation over different network settings determined that the best network architecture was composed by two hidden layers with 400 neurons each. Table II shows the average results of the neural network set out of 1000 playouts. Though the average piece placement time is still low, the number of completed lines is unfortunately much lower that the one achieved by the evolutionary heuristic.

C. The Hybrid Approach

Though the neural networks did not provide a reasonably good number of completed lines, they showed good performance at calculating the probability of each possible placement for a piece to be the most suitable. According to the experimental results, the great majority of feasible placements could be easily discarded because of their low probability assigned by the corresponding neural network. Therefore, a hybrid approach in which the neural network set filters out all the low-valued placements for the next piece is designed. This way, the evolutionary heuristic with simulation T=1 operates only on the few placements *recommended* by the neural networks, considerably reducing the computing time needed for calculating the optimal placement, while preserving the accuracy of a two-step lookahead searching function. Firstly, the neural networks set receives as inputs the piece to place and the reduced version of the game board, and it produces a probability vector $P = [p_1, ..., p_n]$ where each feasible placement is given a probability of being optimal. Then, from all feasible placements, a set of candidate placements $C = \{c_1, ..., c_m\}$, $m \leq n$, is selected

TABLE III

AVERAGE RESULTS FROM THE HYBRID SYSTEM USING SEVERAL VALUES FOR s

s	0.1	0.01	0.001	0.0001	0.00001
Average piece placement time (ms)	0.2376	0.3232	0.4273	0.5518	0.6705
Lines					
Average	37574	98081	188882	313302	523164
Standard deviation	36548	95919	165576	305158	588621
95% confidence interval	[30411, 44738]	[79281, 116882]	[156430, 221336]	[253491, 373113]	[407795, 638534]
Min	314	343	2798	3093	8122
Max	181450	525725	700576	1832349	3658667

and used by the heuristic with simulation T=1, together with the original board, in order to select the optimal placement. Feasible placements are ordered by probability and those of highest probability are chosen one by one until the sum of its probabilities reaches a fixed parameter $s \in [0, 1]$. This way the average number m of candidate placements can be tuned according to s. Note that $s = 0$ selects all of them as candidates, $C = \{c_1, ..., c_n\}$, behaving as the standard simulation T=1 (no placements are filtered out).

Table III shows the results of the hybrid system operating under five different values for s from 100 executions run each. In this case, the results regarding the number completed lines are significantly better than the ones obtained with the previous approaches, specially when $s \leq 0.01$. Though the average piece placement time when $s = 0.00001$ is almost three times the one under $s = 0.1$, the average number of completed lines is almost fourteen times higher. Yet 0.6705 ms is an acceptable time for piece placement in this later case, and the 3658667 lines completed by the optimal produced controller is a remarkable score.

III. CONCLUSIONS AND FUTURE LINES OF RESEARCH

We have presented three controllers for automatically playing Tetris: an evolutionary heuristic, a set of artificial neural networks, and a hybrid system that combines both. The heuristic shows the fastest response time but the average number of completed lines is not very high. The neural network is no match for the heuristic, but it shows good results at classifying future game states as potentially good for the controller. This is used by the hybrid system in order to reduce the amount of possible states that will be evaluated by the two-step lookahead algorithm fed with the evolutionary heuristic. The modularity of this system allows that both the heuristic and the neural networks can be independently tuned. In fact, the configurations presented in this work for both isolated systems perform fairly well. Nevertheless, the hybrid approach provides better results regarding the average number of completed lines with a few increase in computational resources. If either the evolutionary heuristic or neural systems were improved, e.g. exploring different game board features or neural architectures, then the hybrid controller would also provide better results.

Regarding the results achieved, we plan to carry out a comparative study between the presented hybrid system and other successful techniques [12], [14], in order to better explore the advantages of this contribution to the state-of-the-art.

REFERENCES

[1] Tetris Holding. About tetris. http://tetris.com/about-tetris (last accessed 2017-03-07).

[2] S. W. Golomb. *Polyominoes: puzzles, patterns, problems, and packings*. Princeton University Press, 1996.

[3] N. Lundgaard and B. McKee. Reinforcement learning and neural networks for tetris. Technical report, Technical Report, University of Oklahoma, 2006.

[4] C. H. Papadimitriou. Games against nature. *Journal of Computer and System Sciences*, 31(2):288–301, 1985.

[5] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. In *International Computing and Combinatorics Conference*, pages 351–363. Springer, 2003.

[6] H. Burgiel. How to lose at tetris. *The Mathematical Gazette*, 81(491):194–200, 1997.

[7] E. V. Siegel and A. D. Chaffee. Genetically optimizing the speed of programs evolved to play tetris. *Advances in genetic programming*, 2:279–298, 1996.

[8] T. Bousch and J. Mairesse. Asymptotic height optimization for topical ifs, tetris heaps, and the finiteness conjecture. *Journal of the American Mathematical Society*, 15(1):77–111, 2002.

[9] V. F. Farias and B. Van Roy. Tetris: A study of randomized constraint sampling. In *Probabilistic and Randomized Methods for Design Under Uncertainty*, pages 189–201. Springer, 2006.

[10] I. Szita and A. Lörincz. Learning tetris using the noisy cross-entropy method. *Neural computation*, 18(12):2936–2941, 2006.

[11] A. Boumaza. On the evolution of artificial tetris players. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 387–393. IEEE, 2009.

[12] C. Thiery and B. Scherrer. Building controllers for tetris. *Icga Journal*, 32(1):3–11, 2009.

[13] C. E Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer, 1988.

[14] C. Fahey. Tetris. http://www.colinfahey.com/tetris/tetris.html (last accessed 2017-03-14.

[15] N. Böhm, G. Kókai, and S. Mandl. An evolutionary approach to tetris. In *The Sixth Metaheuristics International Conference (MIC2005)*, page 5, 2005.

[16] D. Carr. Applying reinforcement learning to tetris. *Department of Computer Science Rhodes University*, 2005.

[17] X. Chen, H. Wang, W. Wang, Y. Shi, and Y. Gao. Apply ant colony optimization to tetris. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1741–1742. ACM, 2009.

[18] S. Grand, D. Cliff, and A. Malhotra. Creatures: Artificial life autonomous software agents for home entertainment. In *Proceedings of the first international conference on Autonomous agents*, pages 22–29. ACM, 1997.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[20] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[21] M. Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015.

[22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[23] D. Barrios-Rolanía, A. Carrascal, D. Manrique, and J. Ríos. Optimisation with real-coded genetic algorithms based on mathematical morphology. *International Journal of Computer Mathematics*, 80(3):275–293, 2003.

[24] S. Haykin. *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA:, 2009.