Agente inteligente de jogo TETRIS

Frederico Martins Biber Sampaio, Guilherme Brandão Biber Sampaio, Gustavo Pantuza Coelho Pinto

Departamento de Ciências Exatas e Tecnológicas (DCET) – Centro Universitário de Belo Horizonte (UNI-BH)

{fredmbs, guilhermebiber, gustavopantuza}@gmail.com

Resumo. *O trabalho apresenta a elaboração de um conjunto de softwares para agente inteligente capaz de jogar Tetris, incluindo interface gráfica com o usuário e otimização por algoritmos genéticos.*

1. Introdução

Tetris é um jogo eletrônico elaborado em 1985 por Alexey Pajitnov e Dmitry Pavlovsky. A primeira versão foi elaborada em um computador Electronica 60 (Fahey, 2003). O jogo se tornou um clássico, provavelmente um dos jogos mais conhecidos do mundo, e possui várias implementações, vários campeonatos e foi alvo de vários estudos em diversas áreas como, por exemplo, psicologia (Hoff, 2004).

O Tetris é um jogo divertido e amplamente difundido, com uma miríade de aficionados e com características que tornam sua "solução" desafiadora para a elaboração de agentes inteligentes. Nos últimos dez anos, com o crescimento dos estudos em inteligência artificial, observou-se um aumento do número de artigos científicos das áreas de IA - Inteligência Artificial - relacionados com o Tetris.

O Tetris é composto por um tabuleiro (*board*), dividido em linhas e colunas, normalmente, 20 linhas por 10 colunas. Suas divisões são chamadas de células. Um tabuleiro de 20 x 10 possui 200 células. Ao longo do jogo, objetos compostos sempre de quatro peças quadradas, chamados de tetraminos[1], vão surgindo de forma aleatória, mas em localização (célula) de entrada fixa (spawn), e se dirigem para o fundo do tabuleiro, simulando uma queda em gravidade alterada.

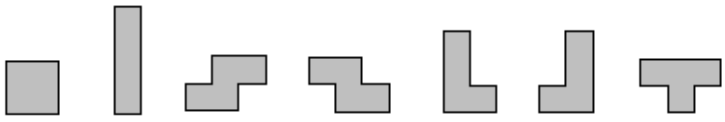

Figura 1 - Tetraminos O, I, S, Z, L, J e T (respectivamente)

[1] Os termos "tetramino" e "tetromino" serão usados de forma indistinta, pois não foi encontrada uma referência objetiva e definitiva em língua portuguesa, ou mesmo em outras línguas estrangeiras. O termo indica elemento composto por quatro componentes (tetra), por isso melhor termo em língua portuguesa é "tetramino".

A gravidade aumenta ao longo do jogo, normalmente de acordo com o nível do jogo (*level*). Os tetraminos formam um empilhamento de peças no fundo do tabuleiro. O jogador pode deslocar o tetramino em queda, antes da sua aterrissagem (*landing*) no empilhamento. Os movimentos podem ser laterais (*right* e *left*), para baixo (*down*) e derrubada do tetramino no empilhamento (*drop*). O jogo termina quando o empilhamento fica muito alto e não permite a saída de novos tetraminos na posição inicial (spawn) no tabuleiro. Para evitar o fim do jogo, qualquer linha que ficar completa é removida (ou erodida) do tabuleiro e o empilhamento reduz.

O Tetris possui características peculiares. Do ponto de vista da IA, seu ambiente é acessível e estocástico. Normalmente, a implementação dos agentes inteligentes, como os do presente trabalho, consideram cada jogada de forma independente da anterior, ou seja, de forma episódica. Mas essa característica pode ser alterada. O jogo pode ser considerado como semidinâmico, pois ele impõe uma gravidade que atrai os tetraminos enquanto o agente delibera sobre a ação a ser tomada. Porém, várias implementações de agentes inteligentes desconsideram a gravidade e, nesse caso, o ambiente é estático. Como o número de peças e os movimentos são bem definidos, o ambiente é discreto. Mesmo o jogo possuindo um único agente, de certa forma, o Tetris é competitivo[2], pois nele são contados pontos usados para ranquear jogadores, como ocorre nas disputas de campeonatos de jogadores humanos[3] e agentes inteligentes computacionais[4].

Estudos sobre eventuais soluções de modelos matemáticos do jogo indicam que ele é, no mínimo, "NP – Difícil" (Demaine, Hohenberger e Liben-Nowell, 2008), sendo que na maioria dos casos estudados a solução é "NP – Completo". Outro problema é sua dinâmica: o Tetris não possui um fim determinado. Estatisticamente, dada uma sequência de tetraminos completamente aleatória, é garantido que sempre existe uma chance do jogo terminar. Dependendo da sequência, o jogo termina de forma rápida. Por exemplo, sequências de peças S e Z (Demaine, Hohenberger e Liben-Nowell, 2008). Também é possível que o jogo se mantenha de forma indefinida. Por exemplo, se ocorrerem apenas peças O em tabuleiros com número de colunas múltiplo de quatro.

Em face das enormes indeterminações do jogo e das dificuldades de se estabelecer uma solução baseada em modelo matemático, as alternativas são heurísticas. Existem várias funções heurísticas que podem ser adotadas por um agente inteligente. Todas avaliam o melhor movimento a ser tomado em relação ao atual tabuleiro e tetramino. Como podem existir diversas funções, existirão diversas variáveis. Ou seja, a equação de avaliação do tabuleiro é "não linear" e o espaço de busca é extremamente grande. Por exemplo, em um tabuleiro padrão de 20 linhas por 10 colunas, o espaço de busca é de (rotações possível dos tetraminos)*(combinações possíveis do tabuleiro). No tabuleiro padrão: $(1 + 2 + 3 + 3 + 4 + 4 + 4) * 2^{199} = 21 * 2^{199} \approx 8{,}03469e{+}59$.

[2]Exemplo de jogo competitivo, envolvendo IA e jogador humano, em http://www.theroundpixel.com/Battle%20Blocks/battleblocks.html.

[3]Exemplos de campeonatos de Tetris em http://tetrischampionship.com e em http://ecstasyoforder.com/#masters.

[4]Exemplos de disputa entre agentes inteligentes de Tetris em http://tetrisapp.appspot.com/ e http://code.google.com/p/tetris-challenge/. Um bom exemplo de disputas em meio acadêmico pode ser encontrada no site Reinforcement Learning Competition, "http://2009.rl-competition.org/tetris.php".

O cenário de uso de múltiplas funções heurísticas é típico de adoção de algoritmos genéticos: encontrar o melhor modelo matemático para equações não lineares. Esse é um dos objetivos do presente trabalho: utilizar algoritmos genéticos para encontrar formas de otimizar o modelo matemático adotado pelo agente inteligente, encontrando a melhor solução possível para a equação não linear que engloba as diversas heurísticas adotadas para o jogo Tetris.

Além dessas motivações, o Tetris representa um cenário controlado para o estudo dos temas propostos, ou seja, estudo de técnicas de Inteligência Artificial e Pesquisa Operacional. Na prática, a adoção de otimização depende de grande expertise na área da aplicação. Na maioria das vezes, a formulação do modelo matemático adequado é o principal desafio, necessitando de profundo conhecimento e domínio da complexidade natural envolvida na área de aplicação. Como o trabalho tem como foco estudos em nível de graduação em Ciência da Computação, considerando aspectos e técnicas computacionais, um modelo mais controlado, como o jogo Tetris, parece mais adequado que modelos reais complexos, pois permite um experimento academicamente mais proveitoso e focado nas etapas de implementação e otimização em si.

Mesmo no modelo simples e controlado do Tetris, como será apresentado ao longo do trabalho, o grupo enfrentou diversas dificuldades práticas, principalmente de ordem computacional, pois os tempos e os esforços de processamento envolvidos nas simulações são altos. Essa dificuldade prática impôs grandes restrições para a obtenção de dados estatísticos para análise.

1.1 Objetivos

O objetivo principal do trabalho é elaborar um agente inteligente capaz de jogar Tetris. Para atingir o objetivo, será elaborado um simulador do jogo Tetris, com interface gráfica e inteligência artificial.

Como a maioria dos jogos eletrônicos, o Tetris é um jogo com grande apelo visual. Essa é a motivação da elaboração da interface gráfica. Além disso, ela será elaborada para facilitar a visualização dos resultados e demonstrações interativas do funcionamento do agente inteligente.

Outro objetivo é otimizar o funcionamento do agente inteligente. Para a otimização do modelo matemático, será elaborado um software de busca utilizando algoritmos genéticos.

Por fim, o trabalho deverá apresentar uma avaliação dos resultados da atuação e da otimização do agente inteligente. Se possível, serão realizadas avaliações comparativas com outros agentes inteligentes encontrados durante as pesquisas.

2. Trabalhos correlatos

Existem vários artigos e exemplos de implementações de agentes inteligentes para o Tetris. Entre os softwares encontrados, destacam-se Fahey (2003), Thiery e Sherrer (2009b), Scherrer, Thiery, Boumaza e Teytaud (2009), El-Ashi (2011), Natan (1999), Johnson (2007) e Li (2011). A principal referência adotada no presente trabalho, citada na maioria dos artigos e softwares encontrados, é Fahey (2003). Sua implementação de IA é baseada em um algoritmo elaborado pelo francês Pierre Dellacherie, em 2003. Além disso, o software utiliza uma abordagem de visão computacional. Nesse caso, um computador atua como jogador. Ele utiliza uma 'webcam' para visualizar a tela do outro computador e identificar o tetramino que surge no jogo. O computador jogador avalia a melhor jogada e envia os comandos do jogo por meio de porta serial.

O algoritmo de Dellacherie usa seis funções heurísticas, cada uma avalia características do tabuleiro. A avaliação final é a somatória da multiplicação dessas funções por seis constantes inteiras, que representam "pesos" de cada heurística na decisão do melhor movimento. As funções heurísticas e os "pesos" são similares às funções de avaliação e aos cromossomas respectivamente.

Os "pesos" usados originalmente, [-1, +1, -1, -1, -4, -1], parecem estabelecidos de forma arbitrária, por isso são bons candidatos para otimização. Mesmo assim, os resultados do algoritmo Dellacherie são significativos. Provavelmente, essa observação da oportunidade de otimização dos "pesos" originalmente propostos incentivou vários estudos posteriores de otimização e aprendizagem aplicados ao Tetris. Praticamente todos os estudos posteriores utilizam funções heurísticas similares, mudando os valores dados aos "pesos". Todos determinaram os novos valores por meios de algoritmos de otimização.

Outra constatação comum nos trabalhos correlatos é a enorme dificuldade de se avaliar os resultados dos jogos. A avaliação da adaptação de um indivíduo (cromossomo) depende do resultado da pontuação de um jogo completo. Contudo, um jogo de Tetris usando IA pode necessitar de milhares de rodadas antes de terminar. Computacionalmente, o tempo de execução de um jogo pode ser enorme. Essa questão afetou profundamente a implementação do presente trabalho e é apresentada na maioria dos artigos, em especial Shahar e West (2010) e Siegel e Chaffee (1996).

Por isso, muitos artigos como, por exemplo, Carr (2005), Zhang, Cai e Nebel (2009), Thiery e Sherrer (2009a) e Romero, Tomes e Yusiong (2010) apresentam uma abordagem de otimização baseada em aprendizagem interativa. Técnicas como "aprendizagem por reforço" permitem mudança dos "pesos" das avaliações heurísticas ao longo da execução de um único jogo, e não ao longo de gerações de jogos.

É relevante destacar uma série de artigos envolvidos em competições acadêmicas como, por exemplo, os artigos de Scherrer *et. al.* (2009), que ganharam o "Reinforcement Learning Competition" de 2008 (Thiery e Sherrer, 2009b). O software elaborado para esse trabalho, o 'mdptetris', provavelmente é um dos mais completos simuladores de Tetris, utilizando diversos fatores heurísticos além dos propostos por Fahey (2003), que também foi usado como referência.

3. Fundamentos teóricos

No geral, as soluções para agentes inteligente capazes de jogar Tetris buscam avaliar as diversas possibilidades de movimento de um determinado tetramino em relação a uma determinada configuração do tabuleiro. A busca avalia cada rotação do tetramino. Para cada rotação, a busca percorre todas as colunas em que o tetramino pode ser colocado.

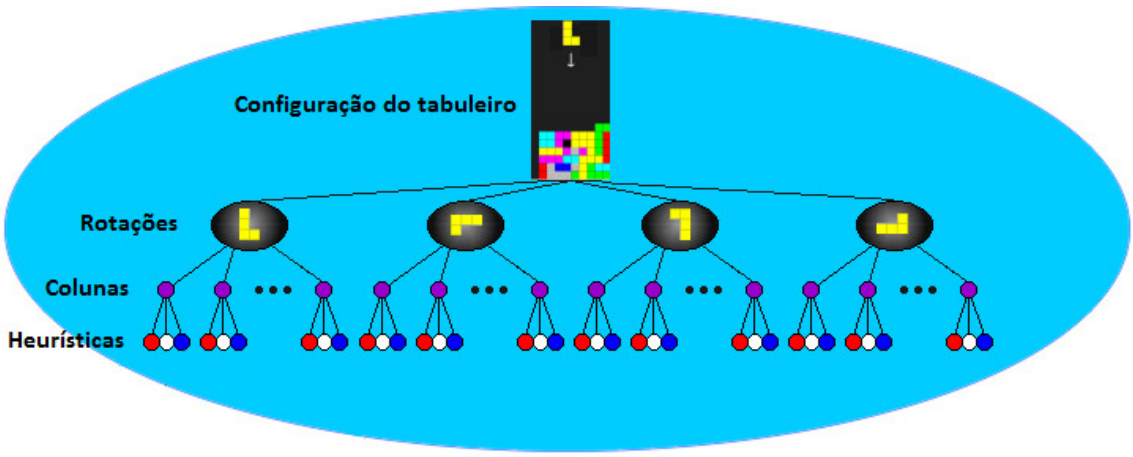

Figura 2 - Busca pelo melhor movimento (Natan, 2009)

As funções heurísticas avaliam a configuração do tabuleiro considerando cada colocação possível do tetramino no empilhamento do jogo. A movimentação escolhida é aquela que obtiver a melhor avaliação heurística do tabuleiro. Ou seja, o algoritmo de escolha do melhor movimento é tipicamente um "*best-first*", ou "algoritmo guloso". O agente inteligente deve repetir essa busca para cada novo tetramino sorteado até que o jogo termine. O jogo deve terminar quando (1) o empilhamento impedir a saída de novos tetraminos, (2) por solicitação do usuário ou (3) por limitação, como limitação do número de tetraminos, limitação de tempo, etc.

As heurísticas usadas são simples, como a altura do empilhamento, o número de "buracos enterrados" (células que são encobertas por linhas), poço (colunas abertas de uma única célula de largura), transições entre células vazias e ocupadas, etc. As avaliações heurísticas adotadas no presente trabalho são:

Tabela 1 - Fatores de avaliação do tabuleiro

Lócus ***i***	**Fator (gene)** ***h(i)***	**Descrição**
0	PileHeight	Altura do empilhamento, em número de linhas.
1	LandingHeight	Altura (linha) da posição final do tetramino no empilhamento.
2	ErodedPieces	Número de peças do tetramino em jogo que forem eliminadas pela remoção de linhas (erosão).
3	ErodedRows	Linhas removidas pelo movimento (erosão).

4	ErodedMetric	Critério original adotado no algoritmo de Pierre Dellacherie. ErodedMetric = ErodedPieces * ErodedRows
5	ColumnTransitions	Somatório do número de transições entre células vazias e ocupadas contadas verticalmente por coluna.
6	ColumnBuriedHoles	Número de células vazias que possuem alguma célula ocupada em uma coluna acima.
7	RowTransitions	Somatório do número de transições entre células vazias e ocupadas contadas horizontalmente por linha.
8	ColumnWells	Número de células vazias, que não possuem colunas ocupadas acima, e que as células imediatamente laterais são ocupadas.

É importante considerar que as paredes laterais e o fundo do tabuleiro representam células ocupadas. O topo do tabuleiro representa células vazias.

Desses fatores, os de número (lócus) 0, 2 e 3 não fazem parte do algoritmo de referência de Pierre Dellacherie (Fahey, 2003). A avaliação da altura do empilhamento foi adicionada, pois percebeu-se que a colocações dos tetraminos O e I nas laterais não mudam a avaliação do tabuleiro e, em muitos casos práticos, o jogo termina com empilhamentos laterais.

O número de peças do tetramino e linhas removidas são variáveis relacionadas com a métrica de erosão, pois "*ErodedMetric = ErodedPieces * ErodedRows*". Apesar de estabelecer uma relação entre as variáveis, a equação final ainda é não linear. O motivo da separação foi tentar avaliar se a métrica (multiplicação) é mais relevante que os fatores individuais da erosão.

Muitos outros fatores podem ser considerados. Praticamente todos os artigos encontrados utilizaram um conjunto diferente de funções heurísticas. Esse é um ponto relevante. Não existe um padrão do conjunto de fatores, nem mesmo os fatores similares são calculados da mesma forma entre as diversas implementações, como apresentado em Scherrer *et. al.* (2009). Contudo, o presente trabalho adotou como referência a implementação de Fahey (2003), assim como a maioria dos outros artigos. Os testes de software foram baseados na igualdade de resultados com o software de referência, mesmo que os algoritmos sejam diferentes, com estruturas de dados diferentes e codificados em linguagem diferente da usada em Fahey (2003). A igualdade não é uma obrigação formal, pois, durante os experimentos, verificou-se que a otimização dos algoritmos genéticos se adapta aos valores retornados pelas funções heurísticas. A igualdade foi estabelecida para possibilitar uma análise comparativa mais direta e precisa.

Cada uma dessas funções heurísticas retorna um valor numérico determinístico em relação ao tabuleiro. Como existem várias funções, cada uma delas representa uma variável na equação de avaliação do tabuleiro. Cada uma dessas variáveis pode ser multiplicada por um fator constante, um termo, que determina o "peso", a importância ou a influência na avaliação final do tabuleiro. Dessa forma é possível estabelecer um modelo matemático para estabelecer a função de tomada de decisão do agente inteligente capaz de jogar Tetris.

3.1 Modelagem matemática

Supondo ***i*** o fator heurístico a ser considerado, ***h*** a função de avaliação para um determinado fator ***i*** em relação ao tabuleiro atual em jogo e ***w*** um vetor com constantes dos termos da equação, que multiplicam o resultado da função ***h***, o modelo matemático de avaliação de um tabuleiro de Tetris é:

$$\max \sum_{i=\min}^{\max} w[i] * h(i)$$

No caso, ***i*** ∈ {PileHeight, LandingHeight, ErodedPieces, ErodedRows, ErodedMetric, ColumnTransitions, ColumnBuriedHoles, RowTransitions, ColumnWells}.

Esse tipo de função de avaliação representa uma equação não linear, pois não existe uma interdependência entre as variáveis ***i***. No caso, cada ocorrência de ***i*** representa uma dimensão, e o espaço de busca é multidimensional. Equações não lineares representam uma aplicação típica de algoritmos genéticos, especificamente para otimização ou mesmo para encontrar soluções desses tipos de equações, se for possível. Devido ao objetivo geral, ao escopo e limitação de tamanho do presente artigo acadêmico, não será realizada uma abordagem profunda sobre a técnica de algoritmos genéticos[5].

3.2 Introdução básica e geral da técnica de algoritmos genéticos

Os Algoritmos Genéticos – AG – são algoritmos de melhorias iterativas, da mesma classe dos algoritmos *Hill-Climbing* e *Simulated Annealing*. Os AG são inspirados na teoria da evolução de Darwin que, resumidamente, estabelece uma regra geral de evolução: ao longo das gerações, os indivíduos mais adaptados ao ambiente sobrevivem e os menos adaptados perecem. Na teoria Darwiniana, cada indivíduo possui pequenas diferenças genéticas que representam vantagens ou desvantagens adaptativas. A diversidade genética é uma realidade prática.

Os algoritmos genéticos consideram que cada indivíduo é representado pelo seu código genético. As mudanças genéticas ocorrem por mutação e por mistura de genes durante a reprodução, ou "crossover". Para definir a adaptação do indivíduo, um AG deve adotar uma função que retorna a adaptação, "*fitness*", de um indivíduo ao seu ambiente. Essa função recebe o código genético do indivíduo e retorna um número que indica sua adaptação. Para simular a evolução, um AG estabelece gerações. Cada geração possui uma população (conjunto de indivíduos). Para cada geração, o AG aplica a avaliação em cada indivíduo e seleciona os mais adaptados. Existem várias formas de seleção de indivíduos entre gerações. Além de excluir indivíduos, o AG aumenta a população por meio de cruzamentos entre indivíduos selecionados (operação de reprodução). Para uma amostragem de indivíduos, o AG aplica a operação de mutação. Essas variabilidades, do ponto de vista matemático, tentam evitar pontos de máximo ou mínimo locais. O fim da evolução pode ser definido por convergência ou por limitação do esforço computacional, tempo ou ciclos de geração.

[5] Existem diversos textos sobre algoritmos genéticos na internet. A título de exemplo, um bom tutorial on-line para introdução aos algoritmos genético pode ser encontrado em "http://www.obitko.com/tutorials/genetic-algorithms/index.php".

3.3 Algoritmos genéticos na otimização do modelo matemático

Relembrando o modelo matemático:

$$\max \sum_{i=\min}^{\max} w[i] * h(i)$$

Nos algoritmos genéticos, ***w*** representa um cromossomo. Cada exemplo numérico, instância, de ***w*** é um genoma (genótipo). Cada fator ***i*** do tabuleiro é um alelo que determina características dos organismos. Um exemplo numérico, instância, de um alelo é um gene. A função ***h*** indica uma avaliação de um fator ambiental específico que o organismo é capaz de lidar. A multiplicação ***w[i] * h(i)*** representa a sua atuação no ambiente (fenótipo). No caso, o organismo é um agente inteligente e o ambiente é o tabuleiro. O agente inteligente irá variar de acordo com seu genoma, ou seja, de acordo com os valores numéricos do vetor ***w*** (instância de ***w***).

Essas características genéticas determinam como um indivíduo irá responder a cada movimento, mas não são suficientes para avaliar sua adaptação ao jogo como um todo. Daí, para representar a função de avaliação do indivíduo (*fitness*), será adotada uma função ***S(w)***, sendo '***w***' o genoma de cada indivíduo e ***S*** é a função que avalia sua pontuação ao longo de um determinado jogo. A seleção dos indivíduos (busca) será baseada no valor de ***S(w)***, ou seja, na sua pontuação no final do jogo.

Para possibilitar comparação e seleção dos melhores indivíduos em uma geração específica, toda a população de indivíduos jogará a mesma sequência de tetraminos previamente sorteados aleatoriamente. É importante destacar que a função de sorteio (*random*) é aleatória, mas possui dispersão homogênea. Ou seja, ao longo do sorteio, a proporção total de cada tipo de tetramino tente a ser muito próxima. A sequência de tetraminos sorteada pode ser alterada entre gerações durante o processo evolutivo.

Para efetuar a evolução, é necessário estabelecer formas de seleção dos indivíduos como, por exemplo, elitismo, ranque (classificação) ou roleta. Além disso, entre as gerações os indivíduos podem se modificar e se reproduzir. Para isso, são aplicados operadores genéticos como, por exemplo, cruzamento genético (*crossover*) e mutação. Um dos focos de análise durante o trabalho foi a escolha dos processos de seleção e dos operadores genéticos mais adequados. Contudo, como será descrito nos próximos capítulos, por causa do enorme tempo necessário para coleta de dados e experimentos, esse tipo de análise não foi suficientemente aprofundada no tempo disponível para o trabalho.

Como existem várias versões de Tetris, algumas das características do jogo são tratadas de formas absolutamente diferentes. Por exemplo, a pontuação. Note-se que a pontuação é fundamental, pois é o fator de seleção dos indivíduos na otimização por algoritmos genéticos. A contagem de pontos adotada é baseada na versão original do jogo da Nintendo[6]. A pontuação depende do número de linhas removidas por movimento. Quanto mais linhas forem removidas em um único movimento, maior é a pontuação: 1 linha = 40 pontos, 2 linhas = 100 pontos, 3 linhas = 300 pontos e 4 linhas = 1200 pontos.

[6] http://tetris.wikia.com/wiki/Scoring

Na versão original, os pontos são multiplicados pelo nível do jogo. Quanto maior o nível, maior é a aceleração da gravidade do jogo (velocidade de descida do tetramino). A contagem ou mudança de nível é outro ponto controverso. No software de avaliação dos agentes inteligentes que calcula ***S(w)***, não foi considerado qualquer nível ou gravidade para o cálculo da pontuação. O motivo é simples: a versão final do software de avaliação executa mais de 1.000 movimentos por segundo, no pior caso apurado[7]. Isso representa um tempo de menos de um microssegundo; tempo extremamente pequeno para representar uma aceleração humanamente perceptível. Na versão de simulação por interface gráfica, que pretende demonstrar visualmente os resultados, a aceleração é verificada e contada na pontuação, inclusive para os agentes inteligentes. O critério de mudança de nível adotado é simples, a cada 20 linhas removidas, aumenta-se um nível. A cada nível, reduz-se 0,05 segundos o tempo de queda (*down delay*), até um limite mínimo de 0,05 segundos. O tempo inicial de queda adotado é de 0,5 segundos. A maioria das características de funcionamento é ampla e facilmente configurável no software elaborado para o trabalho.

4. Metodologia utilizada

4.1 Planejamento

A primeira etapa para a elaboração do agente inteligente foi definir uma linguagem de programação para realizar as simulações e otimizações por meio de algoritmos genéticos.

A linguagem escolhida foi o Python, pois é uma linguagem com muitos recursos e várias bibliotecas, amplamente utilizada em trabalhos científicos e aplicações comerciais. Apesar de ser interpretado, Python é capaz de gerar códigos e executar programas com excelente performance para a maioria das aplicações.

A otimização por algoritmos genéticos utilizou o software PyEvolve. A escolha teve como base critérios de avaliação como estabilidade, documentação, exemplos de aplicações, facilidade de uso, geração de gráficos, quantidade de recursos e utilização concreta em diversos trabalhos científicos.

Após a escolha da linguagem e do software de otimização, foram definidos os requisitos para o software do trabalho: (1) capacidade de expansão de novos recursos, (2) controle estrito dos movimentos e do jogo como um todo, independente do jogador ser humano ou um agente computacional, (3) capacidade de geração de informações para análise, (4) ampla capacidade de configuração das características do software, principalmente recursos de simulação e otimização, (5) código o mais simples e de fácil leitura possível.

4.2 Preparação

No início, pretendia-se aproveitar ao máximo os softwares open-source disponíveis, principalmente para a interface gráfica do Tetris. Contudo, observando-se os diversos softwares de Tetris disponíveis em Python, nenhum deles atendeu aos

[7] O pior caso foi simulado em um computador Pentium III de 1GHz, com Linux, que executou uma média de 4000 sugestões de movimentos por segundo.

requisitos estabelecidos. Então, optou-se por desenvolver um novo software. Para isso, foi elaborado um modelo do software. O diagrama de classes do modelo segue abaixo. Ao longo da implementação, o modelo foi sendo alterado e aperfeiçoado. Como o simulador e a otimização são softwares com grande esforço computacional, a principal preocupação ao longo do desenvolvimento foi aproveitar ao máximo os recursos do computador, principalmente o processador com múltiplos núcleos.

4.3 Modelo do software

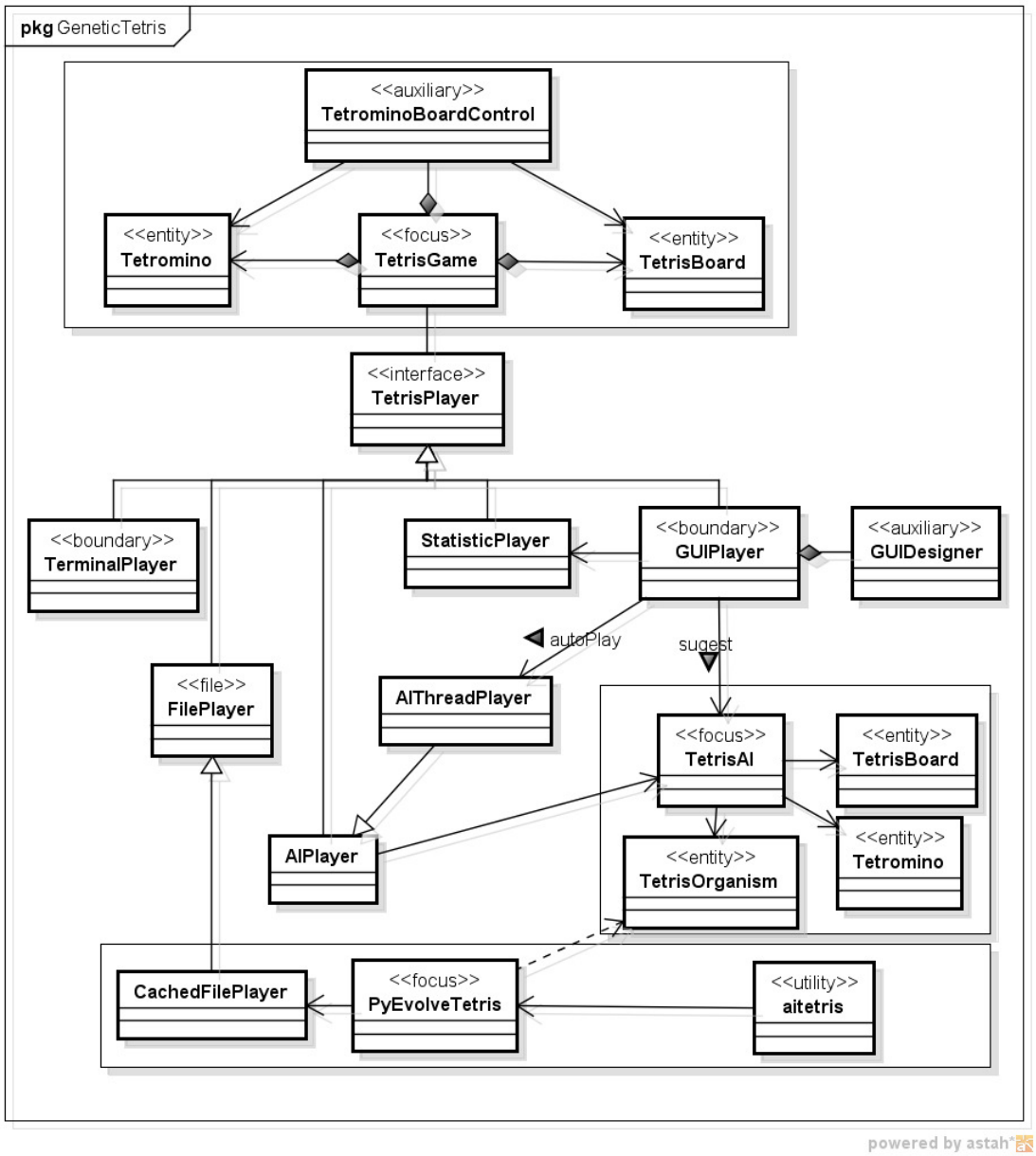

Figura 3 - Diagrama de classes do software elaborado para o trabalho

4.4 Elaboração da versão PYTHON

O software é composto por quatro módulos, representados pelas suas classes principais. (1) TetrisGame, que realiza o controle do jogo, recebe os comandos e envia as notificações de eventos para os players; (2) TetrisPlayer, que aciona os comandos e monitora os eventos do jogo; (3) TetrisAI que implementa o agente inteligente, realizando a busca da melhor jogada, dado um tetramino e um tabuleiro; (4) PyEvolveTetris que realiza a evolução genética, buscando identificar um cromossomo do jogo com melhor avaliação, ou seja, máximo ***S(w)***.

As três classes de entidade do software são: (1) TetrisBoard, que implementa o tabuleiro e as funções heurísticas em Python; (2) Tetramino, que implementa os tetraminos, suas formas, deslocamentos e posicionamento; (3) TetrisOrganism, que implementa o genoma do jogo Tetris e a função de escolha do movimento, ou seja, o somatório de ***w[i]*h(i)***.

Os comandos do jogo são simples: *start*, *move*, *terminate*. Os comandos de movimentação são: *left*, *right*, *down*, *drop*, *rotate_cc*[8], *rotate_ccw*[9]. Os eventos são:

- GAME_START: informa o início do jogo.
- GAME_OVER: informa o fim do jogo, ou por solicitação ou por empilhamento.
- GAME_GRAVITY: informa mudança na gravidade do jogo (*downDelay*).
- TETRAMINO_NEXT: informa o sorteio do próximo tetramino (*lookahead*).
- TETRAMINO_START: informa que o tetramino ativo surgiu no tabuleiro.
- TETRAMINO_MOVE: informa que o tetramino ativo se movimentou.
- TETRAMINO_BLOCKED: informa uma tentativa bloqueada de movimento.
- TETRAMINO_GRAVITY: informa um movimento por gravidade.
- BOARD_CHANGE: informa que o empilhamento do tabuleiro foi alterado.

A principal forma de flexibilidade e extensibilidade do software é a classe TetrisPlayer e suas descendentes. Os players recebem mensagens da classe TetrisGame, configurando os eventos que eles desejam monitorar para executar sua tarefa. Segue-se um resumo das classes de jogadores:

- TerminalPlayer: gera saída em forma de texto do tabuleiro, para acompanhamento em terminal e geração de log. Não é ativo, ou seja, é um observador, não envia comandos para o jogo.
- StatisticPlayer: se registra e recebe eventos para registro e contabilidade de estatísticas. Essa classe é responsável pela contabilidade de pontuação, análise de tempo médio das jogadas, etc. É um observador.
- FilePlayer: se registra no jogo como o "dado" (*setDice*), ou seja a função de seleção da sequência de tetraminos. A sequência é obtida em um arquivo no qual cada caractere (*ASCII*) é um número do tetramino. Isso possibilita que diferentes jogos utilizem a mesma sequência predefinida de tetraminos.

[8] "**cc**" é a redução de "clock wise" ou sentido horário.
[9] "**ccw**" é a redução de "counter clock wise" ou sentido anti-horário

- CachedFilePlayer: similar ao FilePlayer, mas mantém os dados em memória ao longo da execução, para acelerar a performance da simulação.
- AIPlayer: um jogador ativo, que recebe eventos de sorteio de tetramino e envia comandos de movimento do jogo baseados na decisão do agente inteligente elaborado em Python (TetrisAI).
- AIThreadPlayer: jogador ativo, similar ao AIPlayer, mas que funciona em paralelo aos outros programas Python.
- GUIPlayer: jogador que recebe eventos e envia comandos, refletindo os resultados na interface gráfica com o usuário. Utiliza o módulo "pygame".

4.5 Otimização do agente inteligente

Um dos riscos iniciais era a performance da função de avaliação do jogo, ***S(w)***, pois ela realiza milhares de ciclos de busca dos melhores movimentos (figura 2), um para cada tetramino da sequência sorteada, que pode incluir vários milhões de elementos. As primeiras tentativas buscaram usar programação paralela em Python. Apesar de utilizar threads reais, baseadas no sistema operacional e no processador, a implementação "oficial" do Python, CPython, não pode executar operações paralelas no mesmo programa, por limitação do controle de travamento (*lock*) do interpretador[10]. A criação de processos separados também foi tentada, mas não apresentou bons resultados.

O trabalho seguiu as orientações de Lucks (2008): faça o software em Python e, se for necessário resolver algum problema específico de performance, elabore um módulo em mais baixo nível, como em C.

A técnica utilizada para a elaboração do módulo em C usou estruturas de dados nas quais cada célula do tabuleiro e cada peça do tetramino é um "bit". Cada linha do tabuleiro é um número inteiro. O limite de colunas depende do tamanho em bit do número inteiro usado. Porém, essa restrição depende do tipo de inteiro do C utilizado, do sistema operacional e da plataforma de hardware. Como os testes formam executados em sistemas operacionais Linux e Windows de 32 bits e o tabuleiro adotado foi de 10 colunas, foram utilizados números inteiros de 32 bits. Todos os algoritmos foram baseados em operações de *bitwise*; AND, OR, NOT e SHIFT.

Para aproveitar ao máximo os recursos de processador, principalmente os múltiplos núcleos, adotou-se a OpenMP, que é uma biblioteca multiplataforma para programação paralela e memória compartilhada em C/C++ e Fortan. A melhor performance do módulo C foi obtida paralelizando-se o laço (*for*) de avaliação das colunas, dada uma rotação específica. O software mantém entre 6 e 10 threads simultâneas, dependendo da forma (*shape*) do tetramino.

A versão do simulador de IA em Python, *TetrisAI.py*, é capaz de sugerir cerca de 100 sugestões de movimentações por segundo[11]. Cada sugestão de movimentação

[10] Para maiores detalhes sobre "Global Interpreter Lock", recomenda-se o site do CPython: http://docs.python.org/library/threading.html.

[11] O computador dos testes possui processador Pentium 4 Core2Quad Q9450 de 2666 MHz, placa mãe Gigabyte X48-DQ6 e memória DDR2-800 4-4-4-12 T1, ou PC2-6400.

envolveu, na média apurada nos experimentos, 20 execuções da função de avaliação (***Σ(w[i]*h(i)***)). Ou seja, a versão Python foi capaz de executar cerca de 2.000 avaliações por segundo. A versão em C foi capaz de executar cerca de 18.000 sugestões de movimentações por segundo (cerca de 360.000 avaliações por segundo) utilizando um único núcleo do processador. Com a adoção de programação paralela, o mesmo programa atingiu cerca de 44.000 movimentações por segundo (cerca de 880.000 avaliações por segundo), aproveitando 100% dos quatro núcleos disponíveis no computador. Esses resultados apresentaram uma variação média de 15%, ou seja, x ± 15%, dependendo do uso do computador (aplicações concorrentes).

Mesmo operando a 100% da capacidade do processador, nem as aplicações concorrentes nem o computador como um todo ficam "travados", pois, normalmente, a implementação do OpenMP é bem integrada com o escalonamento do sistema operacional que, sempre que possível, utiliza os recursos do processador e outros elementos de hardware.

O programa C registra um módulo Python chamado "aitetris", que atua como uma classe utilitária. O módulo "aitetris" não depende de bibliotecas externas ou extras, é baseado no código fonte do CPython. Caso esteja disponível o OpenMP, como é o caso da maioria dos compiladores atuais, o código resultante utiliza threads.

Normalmente, para instalar o módulo, basta executar o comando "`python setup.py install`", ou apenas "`setup.py install`". Os arquivos fontes do trabalho incluem as configurações necessárias para o MS Visual Studio 2010.

4.6 Características relevantes da simulação adotadas no módulo "aitetris"

Um dos requisitos do software de simulação é possuir controle estrito dos movimentos e do jogo como um todo. Ou seja, o controle do ambiente do jogo simulado deve ser estritamente igual ao ambiente de controle do jogo normal. Esse controle estrito é proporcionado pelo simulador do artigo de referência (Fahey, 2003). A implementação da simulação em Python atendeu esse requisito.

Contudo, o módulo de simulação "aitetris" não considera a posição inicial (spam) do tetramino. No simulador, considera-se que o tetramino já surge na coluna que ele será efetivamente colocado no empilhamento do tabuleiro. Segundo Thiery e Sherrer (2009b), que adotam o mesmo relaxamento, essa característica também ocorre na implementação de diversos simuladores de Tetris.

4.7 Software de otimização por algoritmo genético

O software de otimização, utilizado na maioria dos experimentos, é implementado na classe "PyEvolveTetris", que utiliza o framework de algoritmos genéticos PyEvolve. O software permite configurar os limites (mim, max) dos valores de cada tipo de gene (alelo). Além disso, também é possível configurar se cada gene será tratado com um número inteiro ou como um número real. O número de indivíduos da população, o número de gerações, o taxa de "crosover" e a taxa de mutação também podem ser configurados. Por fim, pode-se configurar se o software troca ou não a sequência de tetraminos usadas na avaliação de cada geração.

A população é gerada automaticamente pelo PyEvolve. Apesar de ser aleatória, a geração inicial tenta espalhar os indivíduos de forma a cobrir ao máximo o espaço de busca. A seleção de indivíduos entre as gerações adotou duas técnicas: (1) elitismo, ou seja, preservação dos indivíduos de maior adaptação (fitness) em cada geração e (2) roleta, no qual os indivíduos de maior adaptação possuem maior chance de serem selecionados. Por padrão, a classe GSimpleGA do PyEvolve gera mutação de 0.2% e "crossover", ou recombinação reprodutiva, de 90%. Como será detalhado no capítulo de experimentos, os valores default da classe GSimpleGA foram modificados para atingir melhores resultados em tempo hábil.

5. Experimentos realizados

O desenvolvimento do módulo "aitetris", codificado em C, viabilizou alguns experimentos. Porém, a elaboração do módulo só ocorreu no final do prazo do trabalho. Mesmo que o software elaborado seja mais rápido que qualquer outra referência encontrada na Internet, o tempo para um jogo de 10.000.000 de tetraminos leva cerca de 200 segundos para ser executado no computador utilizado nos experimentos (±3.33 minutos). Na prática, a maioria dos jogos durante o processo evolutivo demorou muito pouco. Mas, os que foram bem adaptados, demoraram alguns minutos cada. Por exemplo, uma única evolução com população de 1000 indivíduos e 100 gerações demorou cerca de 7 horas ininterruptas e dedicadas de execução. Como a evolução deve ser baseada em uma amostragem muito grande de indivíduos e várias gerações, o tempo para coleta de dados para análises detalhadas é impraticável dentro do prazo para entrega do trabalho. Mesmo assim, foi possível realizar alguns experimentos com resultados relevantes e curiosos.

Praticamente todo artigo relacionado com o tema do presente trabalho apresentou uma melhoria em relação ao algoritmo original de Pierre Dellacherie. Felizmente, o presente trabalho também encontrou otimizações que demonstraram ser mais eficientes que o algoritmo de referência. Porém, é importante destacar que qualquer otimização está relacionada apenas aos "pesos" dados para as funções heurísticas de Pierre Dellacherie. A maioria dos trabalhos pesquisados, inclusive o presente trabalho, adotaram as mesmas heurísticas. As otimizações demonstram apenas a validade do algoritmo original. Além disso, as comparações servem apenas como critério de avaliação dos resultados. Ao final das experiências, percebe-se que, provavelmente, existem outras formas melhores de avaliação. Mas, essa foi a referência adotada para o presente trabalho. Além disso, várias outras heurísticas poderiam ser acrescentadas para aumentar a eficácia do agente inteligente. Nesse caso, a comparação direta seria ainda mais irrelevante.

5.1 A configuração do algoritmo genético

As experiências utilizaram a classe PyEvolveTetris, elaborada especificamente para o presente trabalho, para descobrir genomas mais adaptados para serem adotados nos experimentos seguintes.

O primeiro problema enfrentado foi definir os limites de cada gene, ou seja, sua variação (min, max). Mesmo depois de várias pesquisas, esse tema não foi

adequadamente tratado em nenhum artigo encontrado. Os limites usados foram arbitrários.

Outro problema foi estabelecer a configuração inicial do algoritmo genético.

> *A probabilidade de recombinação, a probabilidade de mutação e o tamanho da população são definidos antes da execução de um algoritmo genético. Infelizmente, não existe uma metodologia que possa definir os melhores valores para esses parâmetros em relação a um problema específico. Por isso, é necessário testar diversas possibilidades e investigar qual delas apresenta os melhores resultados. Normalmente se empregam em algoritmos genéticos, valores nas seguintes faixas: n: 10 a 200, Prec: 0,5 a 1,0 e Pmut: 0,001 a 0,10. (Guimarães, 2007)*

Vários artigos e apresentações (slides) encontrados na Internet propõe regras para escolha e ajustes das configurações do algoritmo genético. Alguns artigos são dedicados a esse tema como, por exemplo, Gotshall e Rylander (2002) e Alzate (2004). No geral, a escolha da configuração dos parâmetros do algoritmo genético envolve execução de várias evoluções para a determinação da convergência[12]. Note-se um gráfico típico de uma evolução na qual ocorre convergência:

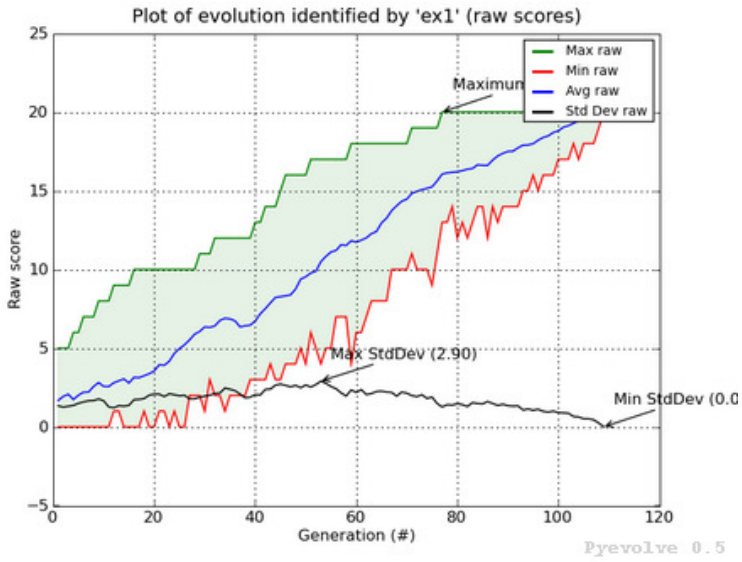

Figura 4 - Gráfico típico de uma evolução com convergência

[12] Apesar de ser relevante, a descrição detalhada das técnicas de configuração de 'algoritmo genéticos' escapa ao escopo do presente trabalho.

O gráfico acima converge, pois os indivíduos mais adaptados atingem o ponto máximo e, ao longo da seleção entre as gerações, os indivíduos menos adaptados também atingem o mesmo ponto de máximo.

Entretanto, é importante notar que, após vários experimentos, dificilmente houve convergência da população de jogos de Tetris para um ponto de máximo. Note-se a seguinte figura com um gráfico de um dos experimentos de evolução, usando população de 5000 indivíduos, 200 gerações e limite de 10.000 tetraminos:

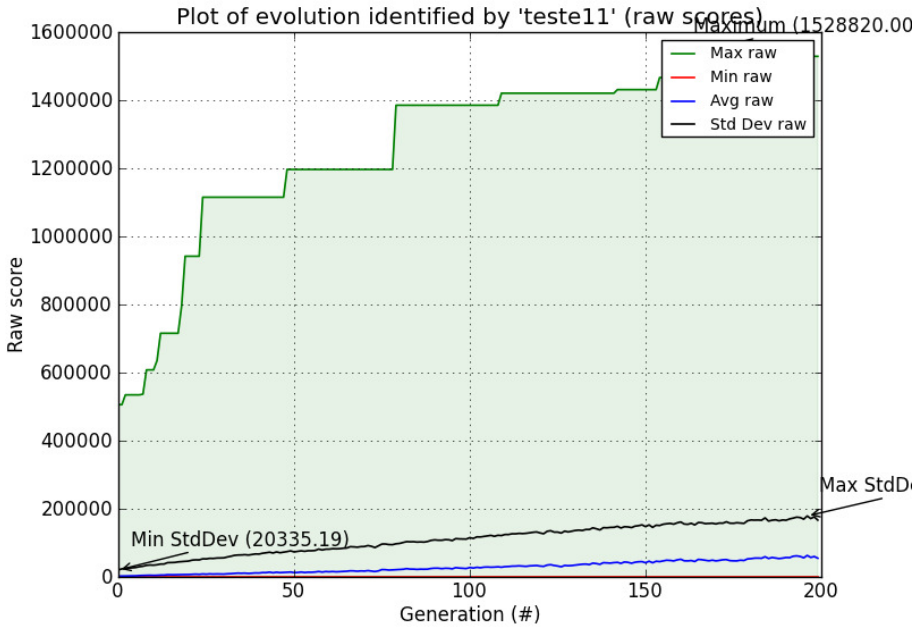

Figura 5 - Gráfico de otimização da função *S(w)* com mutação de 0.2%

De forma geral, a convergência pode ser percebida pela **diferença** entre os maiores valores de avaliação e os menores; quanto menor a diferença, maior é a convergência. Note-se que na figura acima não existe convergência, ou seja, os maiores valores estão se afastando dos menores valores da função de *fitness* (***S(w)***). A divergência é um comportamento natural nos estágios inicias da evolução. Com aumento da mutação de 0.2% para 2% a convergência também não ocorreu, mas a média de "pontos" da população aumentou significativamente e os valores máximos foram atingidos mais rapidamente, mesmo com metade do número de gerações:

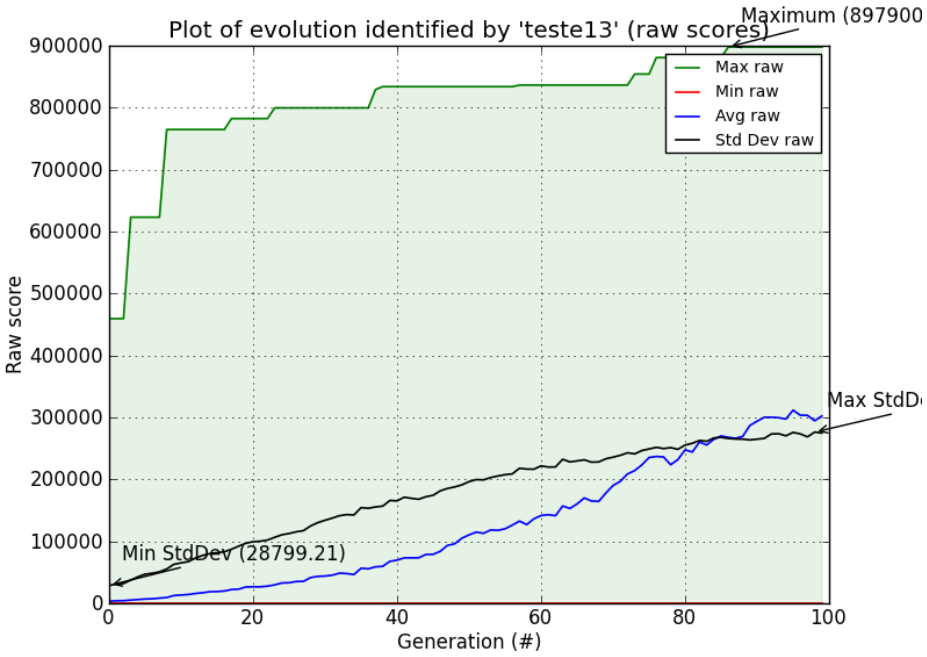

Figura 6 - Gráfico de otimização da função S(w) com mutação de 2%

Como a convergência tende a ser mais afetada pelo número de gerações, tentou-se um novo experimento com 1000 gerações de 100 indivíduos, 10.000 tetraminos, mantendo-se 50% dos indivíduos mais adaptados (elitismo de 50%), mutação de 6% e recombinação (*crossover*) de 90%:

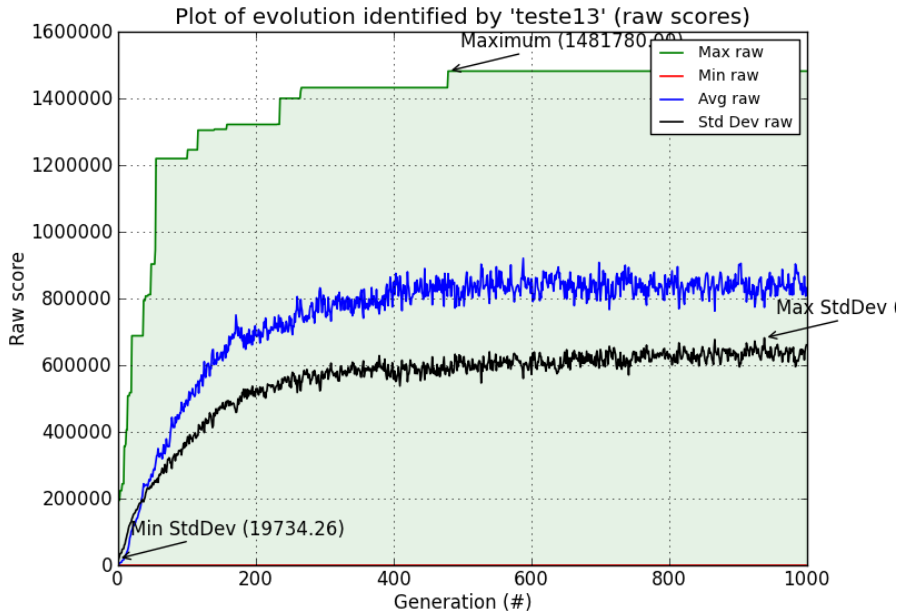

Figura 7 - Gráfico de otimização da função S(w) para 1000 gerações

Observando-se as evoluções, a otimização da função ***S(w)*** é evidente até a geração 400. Depois, continua não havendo convergência. Observa-se que a os valores máximos não se alteram significativamente depois dessa geração. Existe sempre um indivíduo de pontuação zero. Contudo, as pontuações médias também se mantêm estáveis, considerando a mutação de 6%, indicando uma variabilidade grande, porém constante de ***S(w)*** ao longo das gerações posteriores a 400.

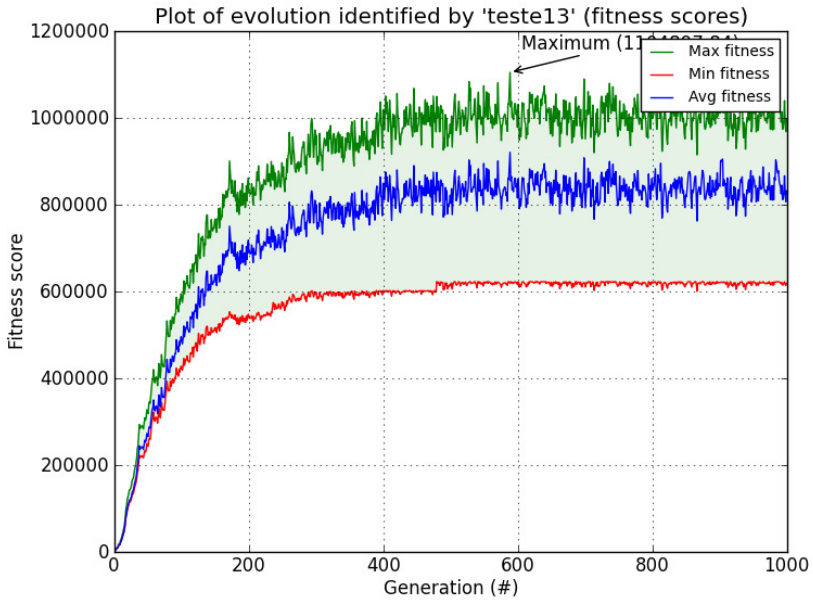

Figura 8 – Variabilidade de *S(w)*

Contudo, a seleção de 50% por elitismo mantém os melhores indivíduos, mas também os tornam parecidos ao longo das gerações. Além disso, mutação de 6% parece muito alta. Para verificar se a variabilidade se manteria alta, porém constante, foram executados outros dois experimentos, de número 14 e 15, com 1000 gerações de 100 indivíduos, 10.000 tetraminos, mutação de 2% e recombinação (*crossover*) de 90%. O experimento 14 adotou 10% de elitismo e genes variando entre -100 e 100 (min, max), enquanto o experimento 15 adotou 5% de elitismo e genes variando entre -10 e 10:

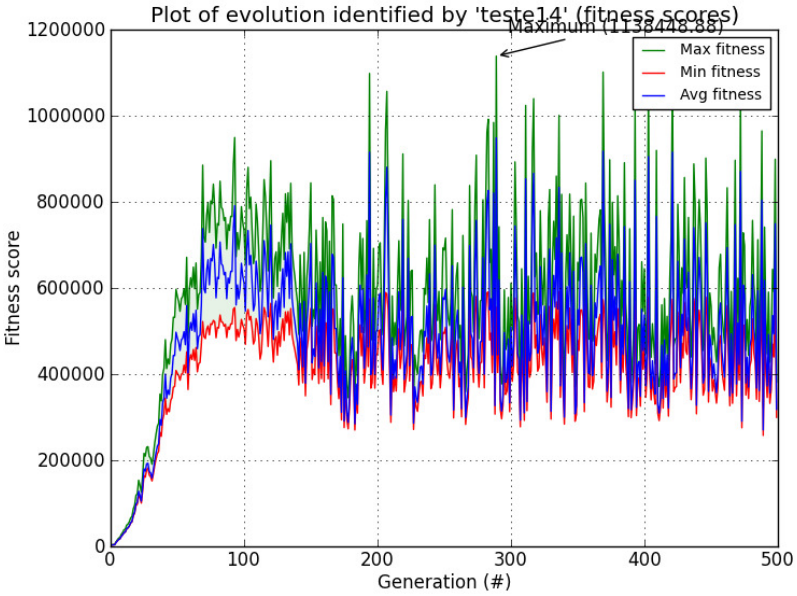

Figura 9 - Variabilidade do experimento 14

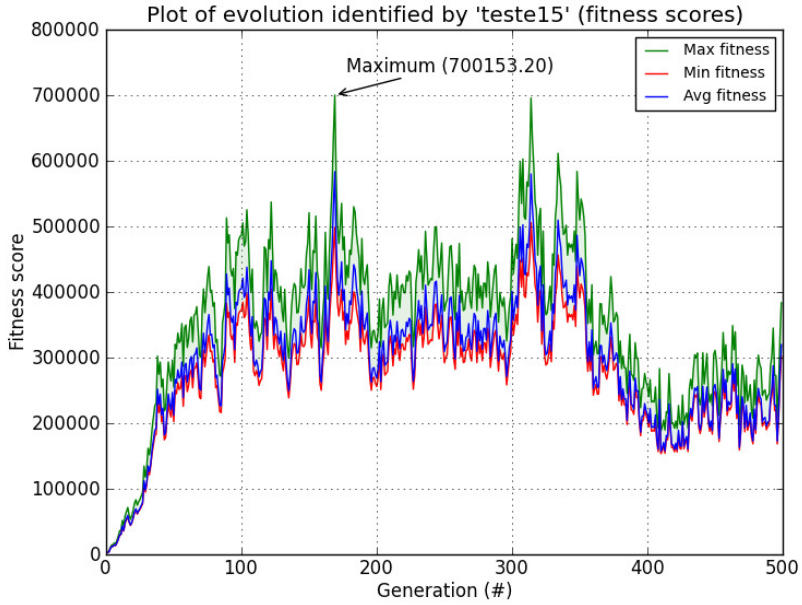

Figura 10 – Variabilidade do experimento 15

Observa-se que a variabilidade se mantém alta, porém quanto menor a preservação de indivíduos por elitismo, maior é a inconstância da variabilidade. O experimento 15 indicou um resultado curioso: uma queda na evolução depois da geração 350.

Como a convergência é um fator fundamental para determinar a adequação do algoritmo genético, decidiu-se realizar um experimento com população de apenas 10 indivíduos, 500 gerações, jogos de 1.000.000 de tetraminos, mutação de 2% e recombinação (*crossover*) de 90%. A execução do teste demorou cerca de 6 horas e terminou por convergência na geração 159. Como foram usados poucos indivíduos, as variabilidades de resultados continuaram altas até o final da evolução.

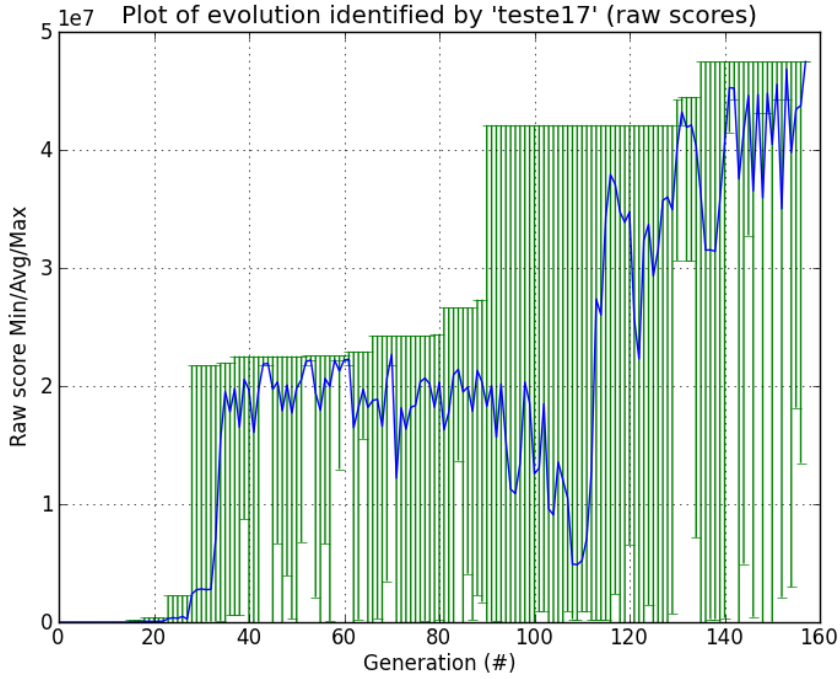

Figura 11 – Exemplo de evolução convergente no Tetris

Mesmo sendo encontrado um cenário de convergência, essa é uma ocorrência muito difícil em populações maiores. Ou seja, uma evolução convergente pode demandar semanas, ou mesmo meses de execução em um único computador. Uma possibilidade de obtenção de resultados em tempo mais rápido seria o uso de processamento distribuído. Essa alternativa foi considerada ao longo de toda implementação, porém nem o tempo e nem o escopo do trabalho permitiram essa abordagem.

O objetivo final da otimização é definir o indivíduo capaz de atingir maiores pontuações. Em todos os experimentos, a pontuação máxima foi atingida entre 200 e 400 gerações. Ou seja, deve-se adotar no mínimo 400 gerações para garantir um resultado satisfatório. O aumento da mutação acelerou o encontro do valor máximo. Baseado nos resultados dos experimentos e nas regras práticas encontradas nas referências bibliográficas, uma mutação de 2% e uma recombinação de 90% parecem adequados. O tamanho da população é um fator crítico, pois afeta o tempo necessário para execução das evoluções. Quanto maior a população, melhor é o resultado do algoritmo genético. Porém, mesmo com populações pequenas, os resultados foram razoáveis. A população mínima recomendada seria entre 100 e 10.000 indivíduos.

As outras observações, principalmente a variabilidade, indicam que o espaço de busca da equação não linear definida no presente trabalho para o jogo de Tetris não tende a um máximo global ou não tende a terminar por convergência. Provavelmente, o espaço de busca esteja repleto de máximos e mínimos locais, caracterizando uma superfície com grande irregularidade.

5.2 Busca de genomas pelo PyEvolve

Ao longo da elaboração do trabalho, foram adotadas evoluções com configurações diversas, pois não existiam dados estatísticos confiáveis para uma decisão sobre população, número de gerações, mutação, taxa de "crossover", percentual de seleção por elitismo e valores limites (min,max) dos genes.

Nos primeiros experimentos, adotou-se população de 1.000 indivíduos, 100 gerações, mutação de 0.2%, "crossover" de 90%, o mesmo jogo (sequência de tetraminos) durante todas as gerações. Foram realizadas várias evoluções, as mais significativas foram:

1. Melhor genoma com todos os genes com números reais entre -100 e 100.
2. Melhor genoma com os genes usados no algoritmo de referência, ou seja, os genomas 0, 2 e 3 zerados. Nesse caso, foram adotados apenas números inteiros entre -5 e 5.
3. Melhor genoma com os genes usados no algoritmo de referência, ou seja, os genomas 0, 2 e 3 zerados. Nesse caso, foram adotados apenas números reais entre -100 e 100.
4. Melhor genoma com os genes usados no algoritmo de referência mais a altura do empilhamento. Nesse caso, foram adotados apenas números inteiros entre -5 e 5.

Outra decisão inicial importante foi adotar uma única sequência aleatória com 10.000.000 tetraminos, pois cada sorteio demorava cerca de 30 segundos, inviabilizando o tempo do experimento. Contudo, essa decisão poderia implicar em identificações de indivíduos "viciados" nessa sequência específica de tetraminos. Nos experimentos, tentou-se verificar o impacto dessa decisão. Por esse motivo, todas as evoluções acima (1 a 4) foram repetidas para comparar os resultados. Nessas novas evoluções, dada a limitação de tempo, adotou-se jogos aleatórios de sequências de apenas 10.000 tetraminos.

5.3 Comparação dos indivíduos

Para realizar essa e as demais a comparações, foi implementada uma classe Python chamada GeneticAnalisys. Basicamente, a classe realiza uma série de jogos com diferentes indivíduos e apresenta o número de vitórias e os melhores resultados. O programa pode ser configurado para usar partidas arquivadas em disco ou sortear durante a execução. O número de partidas por execução é configurável.

As primeiras 4 evoluções utilizaram um exemplo específico de sequência de tetraminos, um arquivo com 10.000.000 números aleatórios. Na comparação, foram

incluídos os melhores indivíduos de cada evolução e todos os genes encontrados em outros trabalhos e implementações similares pesquisadas. Nessa competição o melhor indivíduo foi o encontrado na evolução (1):

```
"01" = [ -0.6903200846016033,-48.397173471721274,  29.57801575198271,
           4.947969428674859,   0.9258831965633374,-24.337463194268338,
         -72.90931022325682,  -93.26927285333721,  -38.827245269520624]
```

Obviamente, como o indivíduo "01" foi evoluído usando uma sequência específica de tetraminos, é de se esperar que seu desempenho seja melhor que os demais nessa mesma sequência. Isso prova o princípio de otimização dos algoritmos genéticos.

A questão é se o desempenho se manterá em outras sequências aleatórias. As comparações foram feitas utilizando-se sequências aleatórias com 10.000.000 de tetraminos. O grande número teve como objetivo verificar a longevidade do indivíduo, ou seja, a capacidade dos indivíduos em se manter no jogo. O indivíduo "01" competiu com o indivíduo "PD" (Pierre Dellacherie), do algoritmo de referência. Em dois experimentos, cada um com 10 partidas, ocorreu empate no número de vitórias entre esses dois indivíduos.

A mesma comparação, dois a dois com o indivíduo "PD", foi realizada entre os melhores indivíduos da evolução (2), (3) e (4), em rodadas de 10 jogos aleatórios. No geral, os genomas encontrados nas evoluções (2) e (3) apresentaram resultados melhores que o genoma "PD".

```
"02" = [ 0.0, -2, 0.0, 0.0, +1, -1, -5, -4, -2])
"03" = [ 0.0,              -56.34441992955155,    0.0,
           0.0,               25.160002295790907, -21.073885962160432,
         -70.94291300255718, -81.7541860917633,   -30.529664695917972]
```

O resultado do genoma da evolução número (4) parece comprovar a relevância de se adotar a altura do empilhamento como um fator genético (lócus 0), pois ele venceu 6 das 10 partidas aleatórias contra o indivíduo de referência.

```
test.addIndividual("04",[ -1, -1, 0.0, 0.0, +1, -1, -4, -3, -2])
test.addIndividual("PD",[0.0, -1, 0.0, 0.0, +1, -1, -1, -4, -1])
```

Como o tempo era curto, para os demais experimentos, adotou-se uma limitação do número de peças das partidas. Na prática, a maioria dos indivíduos atingiu o final da sequência de tetraminos antes de transbordar o empilhamento, mas com pontuações diferentes. Nesse cenário, os testes da relevância do gene de lócus 0 (altura do empilhamento) foi refeito em vários experimentos. Em todos os casos, o indivíduo "04" ganhou da referência.

Porém, adicionados dois indivíduos similares ao de referência, apenas como o gene de lócus 0 com valor de "-0.1" e "-0.5", esses indivíduos quase sempre perderam ou, no máximo, empataram com o genoma original. Ou seja, não é simples, e talvez nem viável, comprovar a relevância de um fator genético novo por comparação com indivíduos já existentes.

A separação dos fatores de eliminação de linhas (*ErodedRows*, lócus 3) e peças do tetramino (*ErodedPieces*, lócus 2) também não comprovou objetiva e

indubitavelmente sua relevância, pois os indivíduos com esses genes não zerados, no geral, praticamente empataram com o indivíduo de referência no número de vitórias em séries de diferentes jogos aleatórios. Além disso, esses indivíduos não demonstraram ser mais adaptados que outros com esses dois genes zerados. Essa separação, do ponto vista matemático, parece apenas expandir (dilatar) o espaço de busca, pois é possível definir uma função que relaciona a métrica de erosão com os seus dois fatores.

5.4 Avaliação de eventuais mudanças do ambiente

Um dos pontos relevantes da pesquisa, apurados ao longo do desenvolvimento do software do trabalho, foi a influência de eventuais mudanças no ambiente do jogo.

Segundo Demaine, Hohenberger e Liben-Nowell (2008), um dos cenários com maior probabilidade de derrota do jogo é a ocorrência apenas de tetraminos S e Z, em tabuleiros cujo número de colunas não é divisível por quatro. Essa é uma mudança do critério de sorteio. Como nesse cenário, o jogo termina rapidamente, todos os genes encontrados foram testados em vários jogos. No geral, nenhum indivíduo se destacou, o desempenho de todos foi lastimável, com retirada mínima de linhas.

Para comprovar a possibilidade de adaptação inclusive nesse tipo de cenário, foram executas duas evoluções, uma com genes usando números inteiros e outra com números reais, cada uma com 100 gerações e população de 1000 indivíduos e sorteio apenas de tetraminos S e Z. Os dois genes encontrados nessas evoluções ganharam 99 de 100 jogos com todas as amostras de indivíduos coletados. Em jogos normais, com sorteio de todos os tetraminos, esses dois genes não ganharam nenhuma partida.

O experimento seguinte tem como objetivo observar o impacto de mudanças na dimensão do tabuleiro. Indo contra a especificação do cenário de S e Z, adotou-se um tabuleiro com 12 colunas, que é um número divisível por 4. Nesse caso, a pontuação média subiu, mas os resultados ainda continuaram muito baixos.

Experimentou-se avaliar a mudança de dimensão do tabuleiro nos cenários de jogos aleatórios com número limitado de tetraminos. Com o aumento de dimensão, maior que 20 X 10, não se percebeu uma mudança notável de desempenho ou dos melhores indivíduos. Contudo, com dimensões menores as mudanças foram radicais, tanto na queda geral de desempenho (pontuação) quanto no índice de vitórias dos indivíduos.

5.5 Curiosidade sobre valores de genes:

A principal curiosidade dos resultados foi encontrar genes bem adaptados, mas com valores inversos ao que parecem. Por exemplo, o gene de avaliação da métrica de erosão (lócus 4), que parece ser um fator positivo na avaliação, ocorreu com valor negativo em um indivíduo capaz de ganhar várias partidas de outros organismos com o gene de lócus 4 positivo. Por exemplo, note-se o cromossomo do indivíduo identificado com “14”:

```
test.addIndividual("14",[0.0, -3, 0.0, 0.0, -1, -2, -5, -5, -2])
test.addIndividual("02",[0.0, -2, 0.0, 0.0, +1, -1, -5, -4, -2])
test.addIndividual("PD",[0.0, -1, 0.0, 0.0, +1, -1, -1, -4, -1])
```

Em dez jogos aleatórios, incluindo outros indivíduos além dos listados acima, o individuo “02” venceu 6 partidas, o “14” venceu 1 e o gene de referência (“PD”) não venceu nenhuma. Ao que parece, a proporção, ou distância, entre os fatores é mais importante que o sinal, positivo ou negativo.

Outro exemplo de gene com sinal trocado ocorreu com um indivíduo que demonstrou grande adaptação. Após dezenas de testes, o indivíduo com maior número de vitórias, vencendo em todas as partidas contra o indivíduo de referência, foi obtido durante a execução dos testes do software, antes dos experimentos controlados:

```
"09" = [ 5.671504912205824, -9.49592892788563,    2.01539413359251,
         -8.37674001734818,   2.9089163698839204, -2.3118950065170125,
         -8.932548918392895, -8.247085669016204,  -2.8006237653192523]
```

O valor mais curioso desse genótipo é o gene de lócus 3 (*ErodedRows*), que avalia a contagem de linhas removidas por movimento. Ele é um número negativo relativamente alto, mesmo em comparação ou proporção aos demais genes. Nesse caso, o aparente contrassenso não parece ter uma explicação trivial. Porém, provavelmente a manutenção de poços ou o adiamento da retirada de linhas amplia a chance de retiradas de 4 linhas pela ocorrência do tetramino I. Esse movimento vale 4 vezes mais que a retirada de 3 linhas e 30 vezes mais que a retirada de uma única linha. Além disso, o indivíduo número “09” se destacou mais nos cenários com limitação de número de tetraminos. Nos experimentos com 10.000.000 de tetraminos aleatórios, o indivíduo ganhou várias partidas, normalmente vencendo rodadas de 10 partidas, mas não teve o mesmo desempenho.

6. Conclusão

O trabalho atingiu seu objetivo de realizar a implementação de um agente inteligente capaz de jogar Tetris. A otimização do agente inteligente por meio de algoritmos genéticos é fácil de ser observada e atingiu desempenho superior ao agente de referência e, certamente, superior a várias outras implementações de agentes inteligentes de Tetris encontradas na Internet. Porém, a falta de tempo impediu a obtenção de dados estatísticos para uma análise mais aprofundada sobre a otimização por algoritmos genéticos.

A elaboração do agente inteligente também poderia incluir mais funções heurísticas. A restrição de tempo também não permitiu essa abordagem. Considerando o impacto da variação dos cenários, o agente inteligente poderia adotar diferentes vetores ***w[i]*** (genomas) para diferentes cenários. Como o algoritmo é tipicamente um “best-first”, uma possibilidade seria manter alguma informação histórica dos movimentos já executados para a determinação dos próximos movimentos. Esse é um campo de pesquisa que não foi encontrado em nenhuma referência bibliográfica.

O Tetris é realmente um jogo difícil, difícil até por aproximação (Demaine, Hohenberger e Liben-Nowell, 2008). As possibilidades, mesmo sendo um modelo controlado e simples, são gigantescas e computacionalmente inviáveis. A primeira observação inevitável é que as técnicas de busca por algoritmos genéticos são relevantes e apresentam excelentes resultados. Todavia, a otimização de equações não lineares,

ainda são computacionalmente desafiadoras, pois podem depender de muito tempo e muito poder de processamento até atingir níveis relevantes de otimização.

A elaboração do trabalho conduziu o grupo de alunos em uma jornada prática e esclarecedora de muitos pontos relevantes relacionados com pesquisa de Inteligência Artificial e otimização de equações não lineares. Mesmo adotando um modelo simples, o grupo percebeu que, na prática, a modelagem do problema, suas heurísticas e a implementação do modelo em forma de software representam boa parte do trabalho.

Inteligência artificial e otimização são temas realmente fascinantes, com enorme potencial para diversas aplicações práticas. Mas, pelo enorme esforço despendido no trabalho para atingir níveis adequados de otimização e critérios científicos de análise, não é de se espantar exemplos como a soluções de jogo de Dama, que demorou 18 anos de pesquisa de uma equipe inteira de cientistas (Schaeffer e Lake, 2006). Aplicações práticas de otimização com algoritmos genéticos podem demandar enormes investimentos e extremo domínio na área de aplicação, incluindo a natural complexidade envolvida. Se o modelo simples do Tetris se tornou complexo e de difícil abordagem no contexto do presente trabalho, imagine-se exemplos reais de uso de algoritmos genéticos como, por exemplo, sistemas de distribuição, logística, produção de energia, operações petrolíferas, operações industriais, etc.

A adoção do Python como linguagem principal demonstrou seu enorme potencial para pesquisas científicas. Provavelmente, com qualquer outra linguagem os problemas de implementação seriam muito piores. A programação em alto-nível, com possibilidade de integração elegante com C, traz segurança e amplia as áreas de aplicação do Python. A curva de aprendizagem da linguagem é rápida e o conjunto de bibliotecas de qualidade disponível em open-source é impressionante.

Outra conclusão inevitável é que linguagens de baixo nível e/ou alta-performance como, por exemplo, C, C++ e Fortan, ainda são instrumentos indispensáveis para a pesquisa de simulação computacional que demandam de alto desempenho, como é o caso de algoritmos genéticos.

Observando os poucos resultado obtidos em face do prazo do presente trabalho, além dos resultados descritos nos diversos outros trabalhos científicos encontrados, conclui-se que o Tetris, além de um excelente exemplo acadêmico, ainda é um campo aberto para pesquisa e abordagens de Inteligência Artificial. Seu ambiente e suas características são surpreendentemente desafiadoras. Por isso, as competições entre agentes inteligentes é uma atividade prática interessante para alunos de graduação em Ciência da Computação (Thiery e Scherrer, 2009b).

O processo de busca (otimização) e análise (competição), mesmo sendo demorado e um pouco tedioso, é instigante e sempre tem uma expectativa, como se a próxima evolução fosse encontrar um genoma “vencedor” capaz de ganhar um torneio e derrotar todos os demais. Na prática, observou-se que alguns indivíduos ganham mais partidas que outros, mas nunca ganha sempre. Esses resultados são condizentes com a dificuldade de convergência observada durante a evolução dos algoritmos genéticos. Além disso, pequenas variações como, por exemplo, nos critérios de sorteio ou nas dimensões do tabuleiro, alteram profundamente o desempenho dos indivíduos. No final

da elaboração deste artigo, a impressão geral é que ainda seriam necessárias muitas pesquisas e experimentos.

Apesar das enormes dificuldades práticas, dos riscos antecipadamente previstos e efetivamente ocorridos, provavelmente o presente trabalho foi o mais divertido e interessante até o atual momento do curso de Ciência da Computação. Ele reuniu algoritmos genéticos, otimização de equações não lineares, modelagem e simulação, projeto de software orientado a objetos, programação paralela, integração de software ente diferentes linguagens, interface gráfica, compatibilidade entre diferentes sistemas operacionais e muito tempo jogando Tetris!

7. Referências

Fahey, Colin P. (2003). **TETRIS**, Disponível em: "http://www.colinfahey.com/tetris/tetris_en.html". Publicado em: 2003. Acessado em: 30/11/2011.

Hoff, Miriam Schifferli and Wechsler, Solange Muglia (2004). **Processo resolutivo do jogo computadorizado Tetris: análise microgenética.** Psicol. Reflex. Crit. vol.17 no.1 Porto Alegre 2004. Disponível em: "http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0102-79722004000100016". Acessado em: 30/11/2011.

Zhang, Dapeng, Cai, Zhongjie and Nebel, Bernhard (2009). **Playing Tetris Using Learning by Imitation.** University of Freiburg, 2009. Diponível em: "http://www.informatik.uni-freiburg.de/~ki/papers/zhang-cai-nebel-gameon10.pdf". Acessado em: 30/11/2011.

Carr, Donald (2005). **Applying reinforcement learning to Tetris.** Rhodes University. May 30, 2005. Disponível em: "http://www.colinfahey.com/tetris/ApplyingReinforcementLearningToTetris_DonaldCarr_RU_AC_ZA.pdf". Acessado em: 30/11/2011.

Romero, Victor II M., Tomes, Leonel L. and Yusiong, John Paul T. (2010). **Tetris Agent Optimization Using Harmony Search Algorithm.** University of the Philippines. 2010. Disponível em: "http://www.ijcsi.org/papers/IJCSI-8-1-22-31.pdf". Acessado em: 30/11/2011.

Demaine, Erik D., Hohenberger, Susan, Liben-Nowell, David (2008). **Tetris is Hard, Even to Approximate.** February 1, 2008. Disponível em: "http://erikdemaine.org/papers/ Tetris_TR2002/paper.pdf". Acessado em: 30/11/2011.

Li, Bai (2011). **Coding a Tetris AI using a Genetic Algorithm.** Publicado em: 27/05/2011. Disponível em: "http://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/". Acessado em: 30/11/2011.

El-Ashi, Islan (2011). **El-Tetris in HTML5. See it in action!** Publicado em 13/08/2011. Diponível em: "http://ielashi.com/el-tetris-in-html5/". Acessado em: 30/11/2011.

Johnson, Damian (2007). **Tetris AI**. Disponível em: "http://www.atagar.com/applets/tetris3D/tetrisAI.php". Acessado em: 30/11/2011.

Natan, Adam (1999). **Learnig to play TETRIS**. Disponível em: "http://www.cs.cornell.edu /boom/1999sp/projects/tetris/". Acessado em: 30/11/2011.

Boumaza, Amine (2009). **On the evolution of artificial Tetris players**. Publicado em: 19/06/2009. Disponível em: "http://hal.archives-ouvertes.fr/docs/00/39/70/45/PDF/ article.pdf". Acessado em: 30/11/2011.

Scherrer, Bruno, Thiery, Christophe, Boumaza, Amine and Teytaud, Olivier (2009). **MdpTetris**. Disponível em: "http://mdptetris.gforge.inria.fr/doc/index.html". Acessado em: 30/11/2011.

Thiery, Christophe and Scherrer, Bruno (2009a). **Improvements On Learning Tetris With Cross-Entrop.** Vandoeuvre-les-Nancy Cedex, France. Publicado em: 22/09/2009. Disponível em: "http://hal.inria.fr/docs/00/41/89/30/PDF/article.pdf". Acessado em: 30/11/2011.

Thiery, Christophe and Scherrer, Bruno (2009b). **Building Controllers For Tetris.** Vandoeuvre-les-Nancy Cedex, France. Publicado em: 22/09/2009. Disponível em: "http://hal.inria.fr/docs/00/41/89/54/PDF/article.pdf". Acessado em: 30/11/2011.

Siegel, Eric V. and Chaffee, Alexander D. (1996). **Genetically Optimizing the Speed of Programs Evolved to Play Tetris.** Advances in Genetic Programming: Volume 2. MIT Press. Outubro de 1996. Dipsonível em: "http://citeseerx.ist.psu.edu/viewdoc/ download?doi=10.1.1.46.4139&rep=rep1&type=pdf". Acessado em: 30/11/2011.

Kistemaker, Steijn (2008). **Cross-Entropy Method for Reinforcement Learning.** Thesis for Bachelor Artificial Intelligence. University of Amsterdam. 27 de Junho de 2008. Disponível em: "http://staff.science.uva.nl/~whiteson/Shimon_Whiteson/ Students_files/Cross-Entropy%20Method%20for%20Reinforcement%20Learning %20-%20Bachelor%20Thesis,%20Steijn%20Kistemaker.pdf". Acessado em: 30/11/2011.

Shahar, Elad and West, Ross (2010). **Evolutionary AI for Tetris.** University of Massachusetts, 2010. Disponível em: "http://www.cs.uml.edu/ecg/pub/uploads/ AIfall10/eshahar_rwest_GATetris.pdf". Acessado em: 30/11/2011.

Lucks, Julius B. (2008). **Open writing projects/Python all a scientist needs.** Pycon 2008. Disponível em: "http://openwetware.org/wiki/Julius_B._Lucks/Projects /Python_All_A_Scientist_Needs". Acessado em: 30/11/2011.

Saha, Amit Kumar (2010). **GAs with a starting point?** Publicado em 06/12/2010. Disponível em: "http://echorand.me/2010/01/06/genetic-algorithms-localsearch/". Acessado em: 30/11/2011.

Schaeffer, Jonathan and Lake, Robert (2006). **Solving the Game of Checkers.** Games of No Chance. MSRI Publications. Vol ume 29, 1996. Disponível em: "http:// library.msri.org/books/Book29/files/schaeffer.pdf". Acessado em: 30/11/2011.

Guimarães, Octávio Rosa de Almeida (2007). **Aplicação de Algoritmos Genéticos na Determinação De Cava Final E Seqüenciamento De Lavra Em Minas A Céu Aberto**. Dissertação de Mestrado apresentada ao Curso de Pós-Graduação em Engenharia Metalúrgica e de Minas da Universidade Federal de Minas Gerais.

Orientadora: Professora Maria de Fátima Gripp. Belo Horizonte, Escola de Engenharia da UFMG, Fevereiro/2007. Disponível em: “http://www.bibliotecadigital.ufmg.br/dspace/bitstream/1843/MAPO-77KNW9/1/oct_vio.pdf”. Acessado em: 04/12/2011.

Gotshall, Stanley and Rylander, Bart (2002). **Optimal Population Size and the Genetic Algorithm.** Disponível em: "http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.2431&rep=rep1&type=pdf&ei=be3bTo7HCcrDgAeL35yuBw&usg=AFQjCNE49Hue2t9H04Hr29qNayELixhcLA". Acessado em: 04/12/2011.

Alzate, Mauricio Orozco (2004). **Optimización de los Parámetros de Control del Algoritmo Genético Simple usando Regresión de Vectores Soporte.** Universidad Nacional de Colombia, Manizales, Caldas. Disponível em: “http://www.manizales.unal.edu.co/gta/signal/morozcoa/orozco04GA.pdf”. Acessado em: 04/12/2011.