UNIVERSIDADE FEDERAL DA GRANDE DOURADOS

ÍTTALO LIMA STELTTER

ALGORITMO GENÉTICO E O SUPER MARIO

DOURADOS
2016

ÍTTALO LIMA STELTTER

ALGORITMO GENÉTICO E O SUPER MARIO

Trabalho de Conclusão de Curso de graduação apresentado para obtenção do título de Bacharel em Sistemas de Informação.
Faculdade de Ciências Exatas e Tecnologia
Universidade Federal da Grande Dourados
Orientador: Prof. M.e Anderson Bessa da Costa

DOURADOS
2016

ÍTTALO LIMA STELTTER

ALGORITMO GENÉTICO E O SUPER MARIO

Trabalho de Conclusão de Curso aprovado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação na Universidade Federal da Grande Dourados, pela comissão formada por:

____

Orientador: Prof. M.e Anderson Bessa da Costa
FACET – UFGD

____

Prof. Dr. Valguima Victoria Viana Aguiar Odakura
FACET - UFGD

____

Prof. Dr. Rodrigo Yoshikawa Oeiras
FACET - UFGD

Dourados, 29 de Abril de 2016.

RESUMO

Mario AI Championship foi uma competição que ocorreu entre os anos de 2009 e 2012, o principal objetivo dos competidores era criar agentes autônomos para controlar o personagem Mario, do famoso jogo Super Mario. O objetivo deste tralho foi implementar um controlador autônomo utilizando algoritmo genético. Veremos um estudo sobre os algoritmos genéticos e sobre a API (*Application Programmer Interface*) fornecida pela competição para os competidores. O resultado foi a modelagem e a implementação do algoritmo genético ao jogo da competição.

Palavras-chave: Algoritmo genético; *Mario AI Championship*.

SUMÁRIO

1. INTRODUÇÃO 3
2. REVISÃO BIBLIOGRÁFICA 4
3. OBJETIVO 7
4. METODOLOGIA 7
4.1. ALGORITMOS GENÉTICOS 7
4.1.1. Cromossomos 9
4.1.2. Função de Avaliação (*Fitness*) 9
4.1.3. Seleção 10
4.1.3.1. *Seleção por Roleta* 10
4.1.3.2. *Seleção por Torneio* 12
4.1.4. Cruzamento (*Crossover*) 14
4.1.5. Mutação 14
4.1.6. Elitismo 15
4.2. MARIO AI CHAMPIONSHIP 15
4.3. REPRESENTAÇÃO DO AG AO *INFINITE* MARIO 18
4.3.1. Representação do cromossomo 18
4.3.2. População Inicial 19
4.3.3. Função *Fitness* – Avaliação 20
4.3.4. Função Seleção 20
4.3.5. Função de Crossover 21
4.3.6. Mutação 21
5. RESULTADOS & DISCUSSÃO 21
6. CONCLUSÃO 27
REFERENCIAS 28

1. INTRODUÇÃO

Super Mario é um jogo de videogame que fez bastante sucesso na era dos jogos de plataforma na década de 90. Jogos de plataformas são jogos em duas dimensões onde o objetivo é “passar” pela fase, alcançando um ponto final. No Super Mario, de forma simplória temos o personagem Mario capaz de correr e pular (além de outras ações possíveis, como segurar o casco, soltar fogo).

Nos últimos anos, uma série de competições com ênfase em IA aplicada a jogos foram executadas em conferências internacionais sobre inteligência artificial, vários deles patrocinados pelo *IEEE Computational Intelligence Society*. Estas competições são baseadas em jogos de mesa clássicos (como Othello, Go, Xadrez) ou vídeo jogos (como Pac-Man, Super Mario Bros e Unreal). Nessas competições, os concorrentes apresentam controladores, chamados de agentes, para o jogo. Esses agentes são codificados através da API construída pelos organizadores das competições.

Entre os anos de 2009 e 2012 foi realizada uma competição chamada de *Mario AI Championship*. A princípio a competição tinha como objetivo criar robôs (*bots*), capazes de controlar o personagem Mario em diferentes fases criadas pelos organizadores da competição. Nesses anos diferentes técnicas conseguiram criar soluções para automatizar o Mario, dentre elas: A*, baseado em regras, programação genética e redes neurais (TOGELIUS; KARAKOVSKIY; BAUMGARTEN, 2010).

Algoritmos genéticos são uma classe particular de algoritmos evolutivos que usam técnicas inspiradas pela biologia evolutiva como hereditariedade, mutação, seleção natural e recombinação. Algoritmos genéticos proveem uma abordagem para aprendizado que é baseada em evolução simulada. Hipóteses em geral são descritas como uma cadeia de bits na qual a interpretação depende da aplicação, embora hipóteses também possam ser descritas como expressões simbólicas ou coleção de programas (MITCHELL, 1998).

Neste trabalho modelamos o problema de passar de fase com o Mario como um problema de otimização e buscamos a solução utilizando algoritmos genéticos.

2. REVISÃO BIBLIOGRÁFICA

Os primeiros estudos sobre Inteligência Artificial (IA) tiveram início na década de 40 marcados pela 2ª Guerra Mundial. Esse fato resultou na necessidade de desenvolver uma tecnologia voltada para análise balística, quebra de códigos e cálculos para projetar a bomba atômica. Surgiam, assim, os primeiros grandes projetos de construção de computadores, assim chamados por serem máquinas utilizadas para fazer cálculos (BARBOSA, 2007).

O primeiro trabalho reconhecido como IA foi realizado por Warren McColoch e Walter Pitts em 1943. Este trabalho fora baseado em três fontes: o conhecimento da fisiologia e a função dos neurônios, uma análise formal da logica proposicional, e a teoria da computação de Turing. Warren McColoch e Walter Pitts propuseram pela primeira vez um modelo de neurônios artificiais (NORVIG; RUSSEL, 2014).

Após o trabalho em 1943, surgiram vários exemplos de trabalhos que hoje podem ser caracterizados como IA, entretanto o trabalho de Alan Turing pode ser considerado o mais emblemático. No ano de 1950, Alan Turing apresentou em seu artigo "*Computing Machinery and Intelligence*" o teste de Turing, aprendizagem de máquina, algoritmos genéticos e aprendizagem por reforço (NORVIG; RUSSEL, 2014).

O termo "IA em jogos" (ou "*Game AI* ") é geralmente usado de forma muito ampla, variando desde a representação e controle de comportamento de personagens não controlados pelo jogador *(nonplayer characters* - NPCs) no jogo à problemas de controle de mais baixo nível que geralmente seriam considerados campo da teoria de controle. Algumas vezes até características de modelagem física e detecção de colisão são também englobadas no rótulo "IA". Embora haja uma conexão quando da movimentação dos personagens, modelagem física é um campo em separado e melhor deixado a cargo de ferramentas específicas (KARLSSON, 2006).

Os primeiros jogos que tentaram implementar uma certa "inteligência" nos personagens controlados pelo computador o faziam de maneiras bastante simples, através de regras implementadas por meio de estruturas do *tipo if-then-else* (LIMA et al., 2012).

Percebeu-se rapidamente a necessidade de melhores técnicas de modelagem e para a representação e controle do comportamento desses personagens, por exemplo, permitindo aos

NPCs ter objetivos e incorporarem um estado que permita progredir em direção dos objetivos (KARLSSON, 2006).

Desenvolvedores de jogos costumam usar um conjunto de técnicas simples na implementação das funcionalidades de IA em jogos, especialmente Máquinas de Estado Finitas (*Finite State Machines* - FSMs) e Máquinas de Estado fuzzy (*Fuzzy Finite State Machine* - FuSMs), que são basicamente um conjunto de estados e transições entre estes, usadas para representar comportamentos (LIMA et al., 2012).

Mesmo com esse pequeno conjunto de técnicas, é possível alcançar resultados bastante satisfatórios. Alguns jogos também fazem uso de árvores de decisões como regras de produção, quando algum tipo de raciocínio sobre o mundo do jogo o faz necessário (KARLSSON, 2006).

Além dessas técnicas comumente utilizadas em IA para jogos, existem algumas técnicas que vêm recebendo um crescente interesse dos desenvolvedores; redes neurais, algoritmos genéticos e redes bayesianas são alguns exemplos deste conjunto de técnicas.

A partir de 1990 surgiram novas técnicas nos jogos eletrônicos, tais como: as máquinas de estado finitos no clássico jogo *Doom (1993),* as redes neurais que foram utilizadas pela primeira vez no jogo *BattleCruiser: 3000AD (1996),* máquinas de estados juntamente com scripts fizeram de Half-life *(1998*) ser considerado o jogo com melhor IA na época. Em 2001 *Black & White* chamou atenção da mídia devido ao fato de as criaturas do jogo aprenderem com as decisões tomadas pelos usuários (SCHWAB, 2009).

Em 1997 Quake II explorou combinações entre algoritmos genéticos e redes neurais, no chamado *NeuralBot*. Este oponente utiliza uma rede neural para controlar suas ações e um algoritmo genético para “treinar” sua rede neural. Isto fez com que o oponente seja totalmente autônomo, sem nenhuma espécie de comportamento pré-programado (CUNHA; GIRAFFA, 2001).

Recentemente, uma série de competições de jogos foram organizadas junto com as principais conferências sobre inteligência computacional, e inteligência artificial para jogos. Nessas competições, os concorrentes são convidados a apresentar seus melhores controladores para um jogo em particular, os controladores em seguida são classificados com base em quão

bem eles jogam o jogo. As competições são baseadas em jogos de tabuleiros populares(como, Go, damas, xadrez)[1] ou vídeo games (como Pac-Man[2], jogos de corridas, Mario).

Jogos de tabuleiros são colocados em disputas controladores *vs* controladores, games como Pac-Man, Mario, jogos de corridas, são testados como os controladores jogam uma determinada fase (TOGELIUS; KARAKOVSKIY; BAUMGARTEN, 2010). Na maioria dessas competições, os concorrentes apresentam controladores que utilizam uma API (*Application Programmer Interface*) construída pelos organizadores da competição. A principal motivação dessas competições é a avaliação comparativa das diferentes técnicas de aplicação dos algoritmos de inteligência artificial.

A API utilizada nesse trabalho é de uma dessas competições, a Mario AI *Championship*, realizada entre os anos de 2009 e 2012 (TOGELIUS et al., 2013). Como o próprio nome diz, é uma competição com foco em AI para jogar Mario Bros da melhor forma possível.

Ao longo dos anos foram definidas várias competições dentro da própria competição, cada uma com seus objetivos específicos, portanto, foi dividida em três categorias principais: *GamePlay* (Jogabilidade), *Learning* (Aprendizagem) e *Level Generation* (Geração de níveis/fases).

A competição usa uma versão modificada do Infinite Mario Bros (PERSSON, 2008), que é um clone de domínio público do clássico jogo de plataforma da Nintendo, o Super Mario Bros.

A jogabilidade do Super Mario Bros consiste em controlar os movimentos do personagem Mario através de um plano bidimensional. Mario pode caminhar e correr para a direita e esquerda, pular e (dependendo do estado) atirar bolas de fogo. A gravidade age sobre o Mario, tornando necessário saltar sobre buracos e obstáculos para avançar na fase. O personagem ainda pode estar em um dos três estados: pequeno, grande (possibilitando destruir alguns blocos) e em um estado grande com fogo (pode disparar bolas de fogo).

O objetivo de cada nível é chegar ao fim da fase, o que significa atravessar da esquerda para a direita todo o cenário. Algumas das metas secundárias incluem recolher o maior número possível de moedas que estão espalhadas por toda fase, terminar o nível no menor tempo e

[1] General Game Playing - Competition. Disponível em: <http://games.stanford.edu/>.
[2] MS Pac-Man Competition. Disponível em:< http://cswww.essex.ac.uk/staff/sml/pacman/PacManContest.html>

juntar o maior número de pontos, o que em parte depende do número de moedas recolhidas, o tempo para terminar e o número de inimigos mortos.

Já existem algumas abordagens que utilizam o conceito de evolução para resolver o problema, entretanto são baseados na programação evolutiva e não algoritmos puramente genéticos. Bojarski, S., & Congdon, C. B. (2010) utilizam programação evolutiva para evoluir um conjunto de regras (PEREZ et al., 2011). Togelius, J., Karakovskiy, S., Koutník, J., & Schmidhuber, J. (2010) por sua vez evoluem uma rede neural.

3. OBJETIVO

O objetivo deste trabalho consiste em implementar um algoritmo genético para automatizarmos o personagem Mario. Modelaremos o jogo como um problema de otimização e iremos por meio de um algoritmo genético alcançar uma solução.

O sistema proposto usa algoritmos genéticos para encontrar uma sequência de ações para obter o melhor resultado possível no final do jogo. Para completar ou tentar completar um nível qualquer em Mario AI é necessário que o personagem Mario execute uma série de ações, este número de ações não é conhecida a priori.

4. METODOLOGIA

Nesta seção serão abordados conceitos fundamentais associados aos algoritmos genéticos e sobre o Mario AI. Na Seção 4.1 explicaremos sobre algoritmos e as técnicas relacionadas: cromossomo, seleção, *crossover*, mutação e elitismo. Na Seção 4.2 veremos uma breve explicação sobre o ambiente fornecido pelo Mario AI *Championship*. Na seção 4.3 é apresenta a modelagem do algoritmo genético ao *Infinite Mario*.

4.1 ALGORITMOS GENÉTICOS

Algoritmos Genéticos (AGs), são métodos de otimização e busca inspirados nos mecanismos de evolução de populações de seres vivos (LACERDA; CARVALHO, 1999). São inspirados no modelo de evolução natural de Charles Darwin, princípio da seleção natural e sobrevivência do mais apto. De acordo com Charles Darwin, “Quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior será sua chance de sobreviver e gerar descendentes”.

Um algoritmo genético pode ser definido como uma técnica utilizada para encontrar soluções aproximadas em problemas de otimização e busca, pertencente à classe de algoritmos evolutivos.

Existem diferentes implementações de algoritmos genéticos com algumas variações, seguindo geralmente a seguinte estrutura: o algoritmo inicia uma população de indivíduos que representam possíveis soluções do problema, após isso entra em um laço de ações, a cada interação desse laço os membros da população são avaliados de acordo com uma função de aptidão, são selecionados alguns membros para gerarem uma nova população que herdará características da população anterior. O laço termina quando for encontrada a solução desejada ou quando atingir o número máximo de gerações determinado no algoritmo.

No Algoritmo 1 é apresentado o esquema geral de um algoritmo genético. Primeiramente, o AG gera uma **população inicial** *P* com *n* indivíduos. Essa população inicial é formada por um conjunto aleatório de **cromossomos**, também chamados de hipóteses, essas hipóteses representam possíveis soluções do problema. Seguindo a execução do algoritmo, esta população é avaliada pela função de aptidão (***Fitness***), função responsável por dar uma nota para cada cromossomo, a nota representa o nível de aptidão de cada indivíduo. A cada interação do laço uma nova população *Ps* é gerada, com base nas hipóteses selecionadas. Após **seleção** são aplicados os operadores de ***crossover*** e **mutação**, esses responsáveis por criar uma nova população. Este processo é repetido até que uma solução satisfatória seja encontrada, ou quando um número de gerações limite for alcançada.

```
AG (Fitness, Limite_fitness, P, r, m)

      Fitness:  função que atribui uma nota de avaliação sobre uma hipótese,
      calcula o nível de aptidão.

      Limite_fitness: um limite especificando o critério de terminação.

      n: número de hipóteses a serem incluídas na população.

      r: a fração da população a ser substituída por Crossover em cada
      etapa.

      m: taxa de mutação.

    Inicializar a população P : Gerar n hipóteses

    Avaliar: para cada (h)hipótese em P, calcule Fitness(h)

    Enquanto [max Fitness(h)] < Limite_fitness faça

      Criar uma nova geração, Ps
```

1. *Selecionar:* selecionar probabilisticamente (1 - r)p membros de P para adicionar em Ps. A probabilidade $Pr(\boldsymbol{h_i})$de selecionar hipóteses $\boldsymbol{h_i}$ a partir de P é dada por

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=i}^{p} Fitness\,(h_i)}$$

2. *Crossover:* selecionar probabilisticamente $\frac{r.p}{2}$ pares de hipóteses de P, de acordo com $Pr(\boldsymbol{h_i})$ dado anteriormente. Para cada par, (h_1, h_2), produzir dois *filhos* aplicando o operador de *Crossover.* Inclui todos os filhos em Ps.
3. *Mutação:* escolher m porcento de membros de Ps com probabilidade uniforme. Para cada membro selecionado, inverter um bit randomicamente na sua representação.
4. *Atualizar:* P, Ps
5. *Avalar*: para cada h em P, calcule Fitness(h)

Retorna a hipótese a partir de P que tem a maior aptidão.

Algoritmo 1. Passo a passo do algoritmo genético. (Baseado em Mitchell, T, 1998)

4.1.1 Cromossomos

AG processa uma população de cromossomos. Um cromossomo é uma estrutura de dados, geralmente um vetor ou uma cadeia de bits, que representa uma possível solução do problema.

A representação cromossomial deve representar adequadamente o problema em questão, traduzindo suas informações para uma maneira viável a ser tratada por um computador (LINDEN, 2006). A mais comum, que também é a representação proposta por Holland (1975) é a representação binária de tamanho fixo, onde cada indivíduo é formado por uma cadeia de bits que podem assumir os valores 0 ou 1. Esta representação possui algumas vantagens: além de ser uma representação compacta, facilita os operadores genéticos *crossover* e mutação.

A escolha de uma representação é arbitrária, e deve ser feita a fim de ser a mais adequada possível a solução do problema.

4.1.2 Função de Avaliação (*Fitness*)

A função *fitness* é a maneira utilizada pelos algoritmos genéticos para determinar a qualidade de um indivíduo como solução do problema em questão (LINDEN, 2006). A função de avaliação e a representação cromossomial estão diretamente ligadas ao problema a ser

abordado – caso esses elementos não sejam boas representações deste problema a solução encontrada pode não ser a solução esperada.

Cada indivíduo da população de soluções é avaliado segundo esta função e recebe uma nota, um valor inteiro, onde quanto maior, mais adaptado é o indivíduo. Esta nota é utilizada para a escolha dos indivíduos que se reproduzirão, através de um método de seleção que favorece a escolha de indivíduos melhor avaliados (LINDEN, 2006). O *fitness* de um cromossomo depende do quão aquele cromossomo está apto para solucionar o problema em questão (MITCHELL, 1998).

4.1.3 Seleção

O operador de seleção é responsável por selecionar cromossomos da população para reprodução, a seleção favorece os indivíduos com maior *fitness*, ou seja, quanto melhor sua aptidão, maior a chance de ser selecionado para se reproduzir. Este mecanismo simula a seleção natural, onde os mais aptos são mais capazes de gerarem descendentes.

Importante destacar que indivíduos menos aptos não podem ser totalmente descartados, a fim de evitar a convergência genética, onde a população de indivíduos se tornam cada vez mais semelhantes entre si, isso destrói a diversidade da população, comprometendo a evolução, evitando uma busca mais ampla pelo espaço de soluções.

Existem alguns métodos para implementar a seleção, onde os mais conhecidos são: seleção por roleta e seleção por torneio.

4.1.3.1 Seleção por Roleta

O método da seleção por roleta faz uma analogia as roletas encontradas nos cassinos. Cada cromossomo recebe uma parte proporcional ao seu *fitness,* em relação à soma total dos *fitness* de todos os cromossomos da população.

Neste método, todos os indivíduos de uma população são avaliados pela função *fitness*, e o resultado dessa avaliação é usado como abertura angular em uma roleta. Sendo assim, indivíduos com maior aptidão teriam um grande ângulo nessa roleta, enquanto indivíduos com menor aptidão teriam ângulos cada vez menores, fazendo com que aqueles que tivessem um maior ângulo, tivessem consequentemente uma maior possibilidade de serem selecionados pela roleta. Após a roleta estar montada, a seleção dos pais é feita através de um sorteio aleatório de um elemento na roleta. A Figura 1 ilustra uma roleta montada, cada cromossomo tem sua porcentagem de chance de ser escolhido.

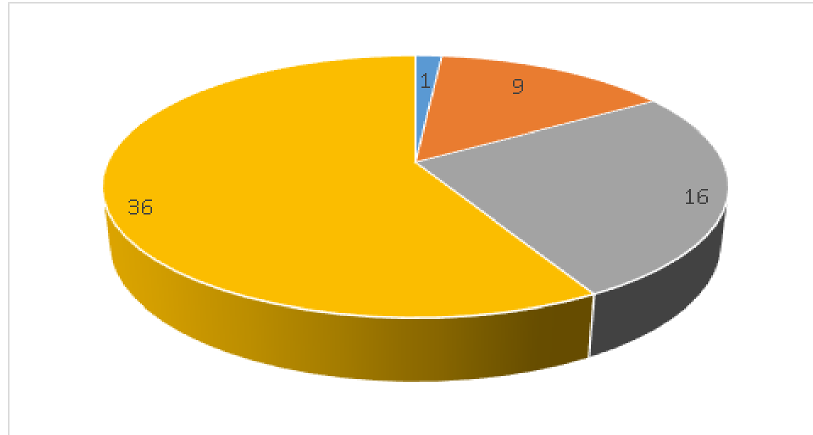

Figura 1. Ilustração da seleção pelo método da roleta, conceito abstrato

Um exemplo para implementação da montagem e seleção pelo algoritmo da roleta, é demonstrado no Algoritmo 2.

```
Seleção por roleta (População, N)
    N: quantidade de indivíduos a ser selecionados
    T recebe a soma de aptidão de todos os indivíduos da População
    Para de i=0 até selecionar N indivíduos faça
        r = valor aleatório entre 0 e T
        S = 0
        Para de j=0 até percorrer toda População faça
            S = S + fitness do indivíduo[j] da População
            Se S >= r então
                Selecione o indivíduo[j]
            Fim se
        Fim do para
    Fim do para
```

Algoritmo 2. Algoritmo da Seleção por Roleta. Fonte: Baseado em Mitchell, T. (1997)

4.1.3.2 Seleção por Torneio

Seleção por torneio é um pouco mais simples. Neste caso, o método consiste em selecionar uma série de indivíduos da população e fazer com que eles entrem em competição direta pelo direito de ser pai, usando como arma a sua avaliação (LINDEN, 2006). São selecionados *N* indivíduos aleatoriamente dentro da população para competir. Uma vez

definidos os competidores, o que possuir a melhor avaliação é selecionado para a aplicação do operador genético. O processo repete-se até selecionar todos os pais desejados para sofrer o processo de *crossover*.

4.1.4 Cruzamento (*Crossover*)

Os operadores de *crossover* e mutação são responsáveis por criar a próxima geração, buscando a evolução e a exploração de regiões desconhecidas do espaço de busca. O cruzamento é aplicado em um par de indivíduos aleatórios, que pertencem à população intermediaria, que foram selecionados no passo anterior, a seleção. O *crossover* tem como objetivo de criar dois novos filhos.

Foram desenvolvidas algumas técnicas de cruzamento que permitem gerar filhos com características herdadas de gerações anteriores e criar populações que busquem a evolução. Os três mais conhecidos, cruzamento simples, cruzamento de múltiplos pontos e o cruzamento uniforme.

O cruzamento simples ou de 1 ponto é o mais utilizado em AG tradicionais, onde temos dois cromossomos pais e ambos são cortados no mesmo ponto, gerando duas cabeças e duas caudas, o ponto de corte é selecionado aleatoriamente. O resultado é a criação de dois filhos, um com a cabeça da mãe e a cauda do pai, outro ao contrário, cabeça do pai e cauda da mãe, esses são os novos cromossomos que vão fazer parte da próxima geração. A Figura 2 demonstra o funcionamento desse método.

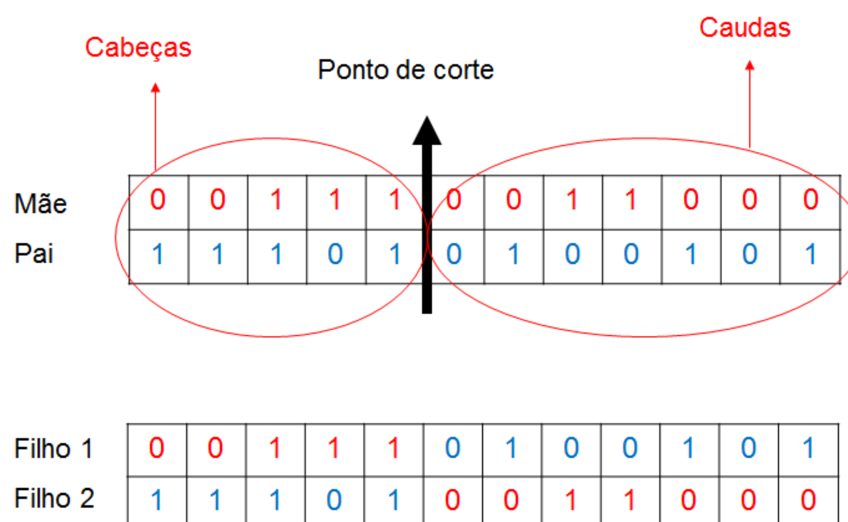

Figura 2. Ilustração do funcionamento do Cruzamento Simples

O cruzamento de múltiplos pontos gera *n* pontos aleatoriamente ao longo da cadeia de bits, criando blocos de tamanhos variados. Esse método resulta em dois filhos de blocos

intercalados dos pais, por exemplo, o primeiro filho utiliza o primeiro bloco da mãe, o segundo do pai, o próximo da mãe e assim por diante até que todos os genes sejam montados, o segundo filho faz o inverso, primeiro bloco genes do pai, segundo da mãe e assim intercalando, entre blocos do pai e da mãe. A Figura 3 demonstra o funcionamento com o valor de $n=3$, ou seja, com 3 pontos de corte.

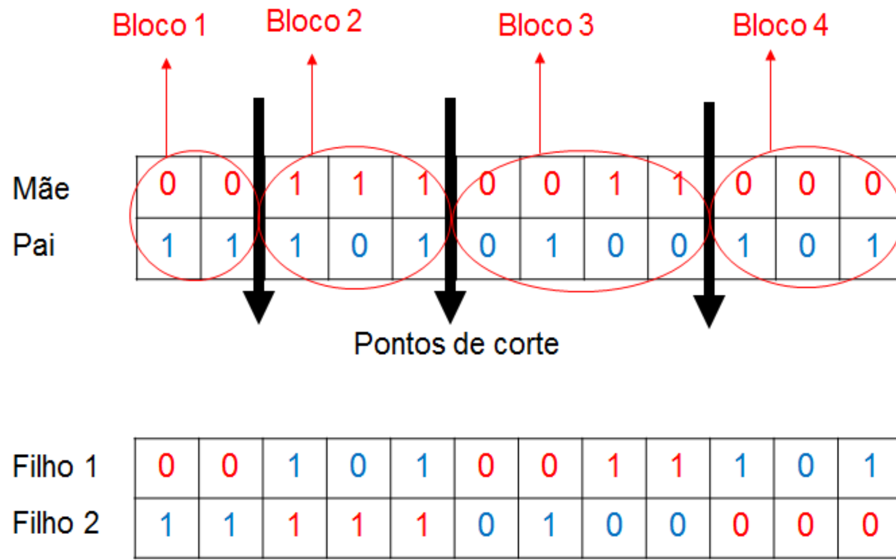

Figura 3. Ilustração do funcionamento do Cruzamento de Múltiplos pontos

O cruzamento uniforme, assim como os anteriores, utiliza um par de cromossomos pais para gerar dois novos filhos, a diferença é que se gera um cromossomo de modo aleatório para cada par de pais selecionados, esse novo cromossomo é conhecido como máscara. A máscara é formada pela a mesma quantidade de bits dos cromossomos pais.

Esta máscara funciona da seguinte maneira: para cada posição da máscara com valor igual a 1 o primeiro filho herdará o bit de mesma posição da mãe, e para cada bit de valor 0 o primeiro filho herdará o bit de mesma posição do pai. O segundo filho é o inverso, se o bit da máscara na posição n for igual a 0 recebe na posição n o bit de mesma posição da mãe, se o bit for igual a 1 recebe o valor do pai. A Figura 4 demonstra a utilização do cromossomo máscara.

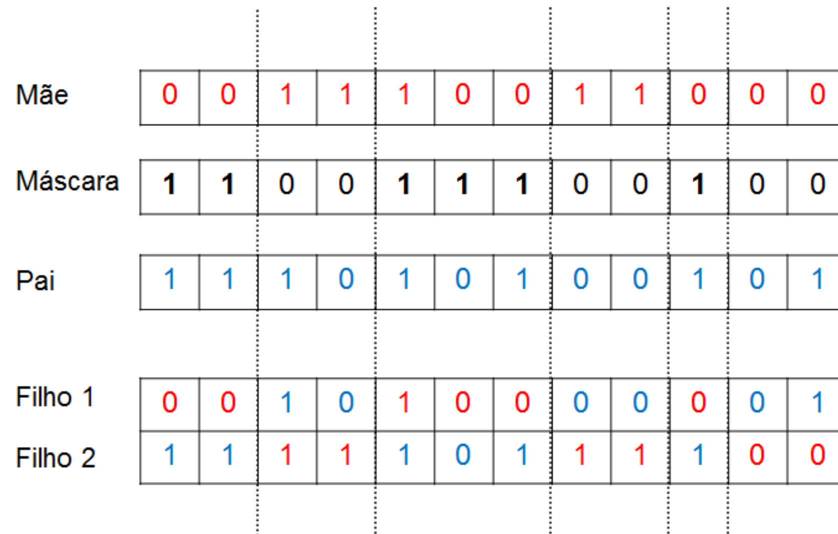

Figura 4. Ilustração do funcionamento do Cruzamento Uniforme

4.1.5 Mutação

A mutação ocorre para que alguns filhos criados possam ter características que não foram herdadas diretamente de seus pais geradores, buscando assim uma maior diversidade na população gerada. Na implementação do operador de mutação, deverá ser definido uma probabilidade de mutação, não é sempre e nem em todos os filhos que ocorre a mutação, essa taxa de probabilidade costuma-se definir um valor baixo, caso contrário a geração perderia muitas características de sua geração anterior.

Esta operação simplesmente modifica aleatoriamente alguma característica de um ou mais genes sobre o filho que é aplicado a mutação, esta troca é importante, pois cria novos valores de características que ainda não existiam, ou que apareciam em pequena quantidade na população. A Figura 5 exemplifica a operação de mutação sobre dois indivíduos, alterando um único bit.

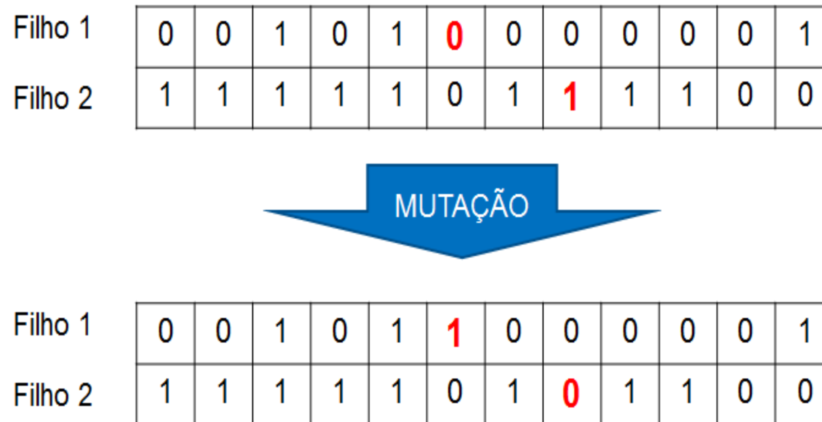

Figura 5. Operador de Mutação

4.1.6 Elitismo

Vale observar que o melhor cromossomo pode ser perdido de uma geração para outra durante os processos de crossover ou de mutação, portanto, é interessante manter o melhor indivíduo da população para a próxima geração sem alterações.

O sistema de elitismo pode ser implementado dentro da seleção, o que garante que o melhor indivíduo até então encontrado seja sempre mantido para a próxima geração, fazendo que suas características sejam passadas adiante e haja sempre a possibilidade de gerar descendentes ainda melhores nas próximas gerações.

4.2 MARIO AI CHAMPIONSHIP

Podemos definir *Mario AI Championship* como uma série de competições com base em um clone do jogo de plataforma semelhante ao Super Mario Bros, e essa competição está dividida em quatro modalidades: *Gameplay*, *Learning*, *Turing Test* e *Level Generation*.

A API do *Mario AI Championship* foi desenvolvida baseada no *Infinite Mario Bros* (PERSSON, 2008), um clone em Java da plataforma do clássico jogo *Super Mario Bros*. O objetivo central do *Infinite Mario Bros*, como do Super Mario Bros, é orientar o personagem Mario do início até o fim em um mundo bidimensional enquanto coleta moedas e supera obstáculos como buracos e inimigos espalhados pela fase. A vantagem do *Infinite Mario Bros* é a possibilidade de gerar diferentes tipos de níveis. Esses níveis são criados a partir de parâmetros como quantidade de obstáculos, quantidade de inimigos e tamanho da fase, a palavra "*Infinite*" do seu título vem dessa característica. A Figura 6 apresenta uma visão ampla de uma fase criada pela API.

Figura 6. Representação de uma fase criada pelo Infinite Mario Bros - Fonte: Karakovskiy, S., & Togelius, J. (2012)

O principal objetivo de cada nível é para chegar ao fim, o que significa atravessar a fase da esquerda para a direita. Metas auxiliares incluem recolher o maior número possível de moedas que estão espalhadas por todo o nível, terminar o nível o mais rápido possível, e recolher o maior número de pontos, o que, em parte, depende do número de moedas recolhidas e inimigos mortos.

Para complicar, existem buracos que o Mario deve evitar e ainda inimigos que se movem pela fase. Se o personagem cair em um buraco, ele "morre". Se ele toca um inimigo, ele se machuca. Machucar significa que pode perder uma vida, se ele está atualmente no menor estado, se não for caso, ele diminui um estado.

O personagem Mario pode estar em três diferentes estados, como ilustrado na Figura 7, da esquerda para a direita, estado de fogo, grande, e pequeno. No *Infinite Mario Bros* o Mario sempre começa no estado de fogo. Quando o personagem é atingido por algum inimigo ele recua um estado, do estado de fogo volta pra grande, e de grande passa pra pequeno, se for atingido em estado pequeno ele morre e finaliza o jogo. No estado de fogo o Mario tem o poder de soltar bolas de fogo para atingir inimigos.

Figura 7 Estados que o personagem Mario pode assumir no jogo. - Fonte: Claessens, R. (2012).

Há diferentes tipos de inimigos no jogo, que estão representados na Figura 8. Da esquerda para direita, *Goomba*, *Winged Goomba*, o *Red Koopa*, o *Winged Red Koopa*, G*reen Koopa*, W*inged Green Koopa*, *Spiky*, o *Winged Skipy*, o *Piranha Flower* e o *Bullet Bill*. O Spiky não pode ser morto pelo jogador, todos os outros inimigos podem ser mortos por bolas de fogo

soltas pelo personagem em seu estado de fogo. Os inimigos podem ser mortos também se o Mario saltar sobre suas cabeças.

Figura 8 Inimigos do Infinite Mario Bros. - Fonte: (CLAESSENS, 2012)

A fim de executar um concurso baseado no jogo *Infinite Mario Bros*, os organizadores tiveram que fazer algumas modificações. Modificações no modo de execução, no modo de entrada da ação, na velocidade de execução, e nos retornos. A cada intervalo de tempo, o jogo fornece informações sobre o que está acontecendo no momento no ambiente. Isto inclui uma matriz (Figura 9), que descreve o mundo ao seu redor, como a localização do Mario, o local e o tipo de inimigos. Também retorna valores de diferentes estatísticas, tais como quantidade de moeda recolhidas, inimigos mortos, o estado de saúde do Mario (Fogo, Grande ou Pequeno).

Figura 9 Matriz retornada pelo sensor da API. Posição onde o Mario pode ir retorna 0, posição onde exite blocos ou inimigos retorna 1.No exemplo o sensor está configurado com o tamanho 7x7, mas pode ser configurado uma matriz de até 22x22. - Fonte: (TOGELIUS et al., 2012)

Ao fim da execução de todos os movimentos, a API fornece como retorno várias informações de como o personagem se comportou durante a execução. São os retornos:

- *distancePassedPhys* - distância percorrida pelo Mario;
- *flowersDevoured* - quantidade de flores recolhida pelo personagem;

- *killsByFire* - quantidade de inimigos mortos pelo fogo do Mario;
- *killsByShell* - quantidade de inimigos mortos por cascos;
- *killsByStomp* - quantidade de inimigos mortos por pulos executados pelo Mario na cabeça dos inimigos.
- *killsTotal* - quantidade total de inimigos mortos na fase;
- *marioMode* - qual o modo se encontra o Mario no final da execução, entre, Fogo, Grande, Pequeno e morto.
- *marioStatus* - retorna se a fase foi completada ou não;
- *mushroomsDevoured* - quantidade de cogumelos devorados, inclui o cogumelo de vida (verde), e o cogumelo de bônus (vermelho);
- *greenMushroomsDevoured* - somente a quantidade de cogumelos verdes;
- *coinsGained* - quantidade de moedas recolhidas;
- *timeLeft* - retorna quanto tempo sobrou no fim da execução caso o personagem tenha chegado ao fim da fase.

4.3 REPRESENTAÇÃO DO AG AO *INFINITE MARIO*

Esta seção apresenta informações de como foi modelado o algoritmo genético ao *Infinite Mario*. Apresenta decisões de como representamos o cromossomo, de como criamos a população inicial e dos modos de seleção e *crossover* selecionados.

4.3.1 Representação do cromossomo

Para completar ou tentar completar um nível qualquer em Mario AI é necessário executar uma série de ações a serem executadas pelo personagem Mario, este número de ações não é conhecida a priori.

Este aspecto influencia no estabelecimento do tamanho dos cromossomos, uma vez que, neste caso, cada quantidade N de bits no cromossomo representa uma ação ou movimento que o Mario tem que realizar em um instante. Por isso, considerando-se o tamanho do nível, tal que seja suficientemente grande de cromossomo para completar o nível.

A API do jogo executa 15 movimentos por segundo, e uma fase tem o tempo definido de 200 segundos. Assim, durante o período de 200 segundos, são necessários 3.000 movimentos. A API trata cada movimento como um vetor booleano de tamanho 6. Esse tamanho é a quantidade de botões que podem ser apertados em jogo. Os 6 botões são: direita,

esquerda, para cima, para baixo, botão para pular e botão para correr ou soltar fogo. Com cada movimento sendo representado por vetor de 6 bits, temos então 64 movimentos diferentes que podem ser executados pelo jogo ($2^6 = 64$).

Precisamos de um vetor com 3.000 movimentos, e cada movimento corresponde uma vetor de tamanho 6 bits, logo nosso cromossomo será representado por um vetor booleano de 18.000 posições. Com isso podemos gerar $2^{18.000}$ diferentes cromossomos, um valor aproximado de $3{,}466 \times 10^{5.418}$.

4.3.2 População Inicial

Para iniciar a população decidimos criar os cromossomos da forma mais simples possível: randômico. A função de iniciar população recebe o número de indivíduos que deve conter na população e gera de forma totalmente randômica cada bit do cromossomo, ao final retorna toda a população.

Porém, depois dos primeiros testes com o algoritmo, percebemos uma grande quantidade de movimentos que não evolui o personagem de lugar, por exemplo, movimentos que executam ir para direita e ir para esquerda ao mesmo tempo. Para melhorar a população inicial, criamos um gerador de população que contenham uma seleção definida de movimentos. São selecionados 3.000 movimentos aleatório entre os movimentos pré-selecionados. Os movimentos escolhidos são representados na Quadro 1. Para criar essa tabela levamos em consideração quais movimentos melhor progredia o Mario pela fase.

MOVIMENTOS	ESQUERDA	DIREITA	PARA BAIXO	PULO	CORRER / FOGO	PARA CIMA
1	0	0	0	1	0	0
2	0	0	0	1	1	0
3	0	0	1	1	0	0
4	0	0	1	1	1	0
5	0	1	0	0	0	0
6	0	1	0	1	0	0
7	0	1	1	0	1	0
8	0	1	1	1	1	0

Quadro 1. Movimentos selecionados para criar população inicial.

Para manter a diversidade de movimentos na população inicial, decidimos que metade da criação seria totalmente randômica com todos os movimentos possíveis, e a outra metade dos indivíduos somente com os movimentos pré-selecionados da Quadro 1.

4.3.3 Função *Fitness* - Avaliação

Nos primeiros testes utilizamos o cálculo da função *fitness* levando em conta somente a distância percorrida pelo Mario, logo percebemos a ineficiência desse método, pois apesar de criar diferentes indivíduos a avaliação retornava o mesmo valor. Esse método não diferenciava um conjunto de ações que chegava ao mesmo ponto vivo ou morto, e se vivo não diferenciava qual o estado do personagem no fim da execução dos 3.000 movimentos.

Pensando em resolver esse problema, utilizamos mais variáveis que a API retorna ao final da execução, valores como o estado do personagem, quantos inimigos mortos, quantas moedas coletadas, quantidade de dano que recebeu de inimigos, bônus por ter chegado ao final da fase e caso termine a fase, quanto tempo sobrou.

Depois de alguns testes o cálculo da avaliação foi definida como:

Fitness = distancePassedPhys + 50 marioMode + killsTotal + coinsGained + killsByFire - collisionsWithCreatures + 500 * marioStatus + timeLeft*

- *distancePassedPhys* - distância percorrida, valores entre 0 e 256;
- *marioMode* - em que modo o Mario está, Morto retorno -1, Pequeno retorno 0, Grande retorno 1, Fogo retorno 2. O peso de valor 50 foi utilizado para balancear, pois os valores entre -1 e 2 pouco diferencia perante as outras variáveis;
- *killsTotal* - quantidade de inimigos mortos;
- *killsByFire* - quantidade de inimigos mortos por fogo;
- *coinsGained* - quantidade de moedas coletadas;
- *collisionsWithCreatures* - quantidade de dano recebido por inimigos;
- *marioStatus* - retorna se o Mario chegou ou não ao final fase, retorno 0 caso não tenha ganhado e retorno 1 caso tenha vencido;
- *timeLeft* - tempo restante da fase caso ele tenha vencido;

4.3.4 Função Seleção

Na função de seleção decidimos pela implementação da seleção por roleta. Não escolhemos a seleção por torneio por não conhecer a quantidade ideal de pais a serem

selecionados para competir. A seleção por roleta não depende de maiores configurações. O próprio algoritmo se comporta garantindo uma seleção proporcional ao seu nível de aptidão.

4.3.5 Função de *Crossover*

Primeiramente foi implementado o *crossover* simples de um único ponto, mas percebemos que a variedade dos filhos gerados ficou bem baixa, a população criada tinha muita semelhança entre os indivíduos da população.

Para resolver o problema implementamos o *crossover* por máscara, e de duas formas. Uma que gera a máscara do mesmo tamanho do cromossomo, e diferencia os filhos gerados bit a bit. O outro método foi gerar uma máscara de tamanho 3.000, esse tamanho representa a quantidade de movimentos, então os filhos gerados herdavam o movimento inteiro dos pais. A escolha entre um método ou outro ficou randômico com iguais chances.

Com os pais selecionados na operação anterior não temos quais casais vão gerar filhos, a formação dos casais aconteceu de forma randômica entre os pais selecionados, o único impedimento é que o casal de pais não podem ser idênticos, pois, se forem idênticos vão gerar dois filhos iguais, isso também contribui para manter a diversidade.

4.3.6 Mutação

A mutação foi implementada de forma que recebe a porcentagem de bits que serão alterados no cromossomo, e recebe a quantidade de cromossomos criados pela mutação.

A escolha dos indivíduos a sofrerem mutação é de forma randômica com todos indivíduos com iguais chances.

5. RESULTADOS & DISCUSSÃO

Após a implementação precisávamos encontrar valores como: a quantidade de indivíduos criados na população inicial, a quantidade de pais selecionados pela função da seleção, a quantidade de filhos que esses pais devem gerar e a quantidade de indivíduos gerados pela mutação.

Os testes foram executados com duas condições de parada, alcançar o objetivo, ou seja, completar a fase, ou completar 1000 gerações de evolução. Os testes serão representados

por gráficos, com valores do *fitness* do melhor indivíduo da geração e da média da população na geração.

Teste 1 – Configuração:

- População inicial: 100 indivíduos
- Quantidade de pais selecionados: 5
- Quantidade de indivíduos criados pelos pais: 20
- Quantidade de indivíduos mutantes: 2

Resultado Teste 1

Tempo de execução: 48 minutos e 46 segundos

Chegou ao fim da fase? NÃO

Após executar 973 gerações, não foi encontrado solução e a população convergiu, ou seja, todos os pais selecionados tinham o mesmo cromossomo, isso impossibilita a evolução da população, pois todos os filhos gerados são idênticos aos pais. Avaliando o Gráfico 1, percebemos a proximidade da média da população com o melhor indivíduo, essa proximidade é prejudicial para a diversidade da população, o que contribuiu para a convergência.

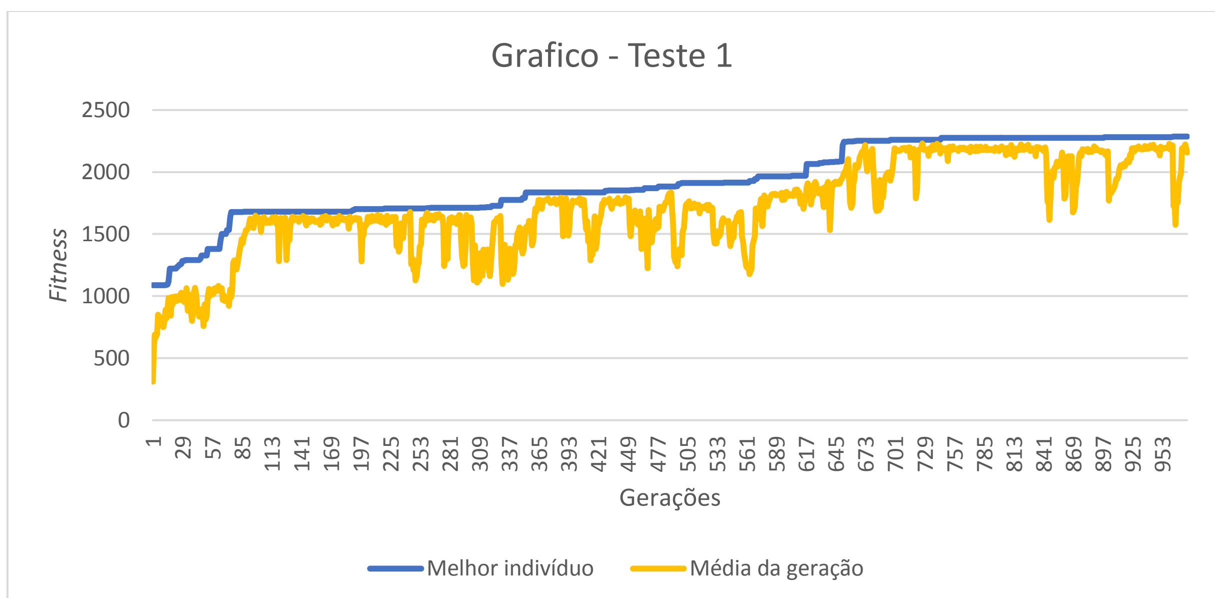

Gráfico 1Representação da evolução do Teste 1

Teste 2 – Configuração:

- População inicial: 200 indivíduos
- Quantidade de pais selecionados: 10
- Quantidade de indivíduos criados pelos pais: 100
- Quantidade de indivíduos mutantes: 10

Resultado Teste 2

Tempo de execução: 69 minutos e 29 segundos

Chegou ao fim da fase? NÃO

No Teste 2, representado pelo Gráfico 2, o algoritmo executou as 1000 gerações, e não chegou ao fim da fase, mas evoluiu melhor que o Teste 1, chegando ao valor do *Fitness* de 2969 contra 2285 do Teste 1. A distância entre o melhor e a média também melhorou, isso pode ter relação com a maior diversidade dentro de cada população criada.

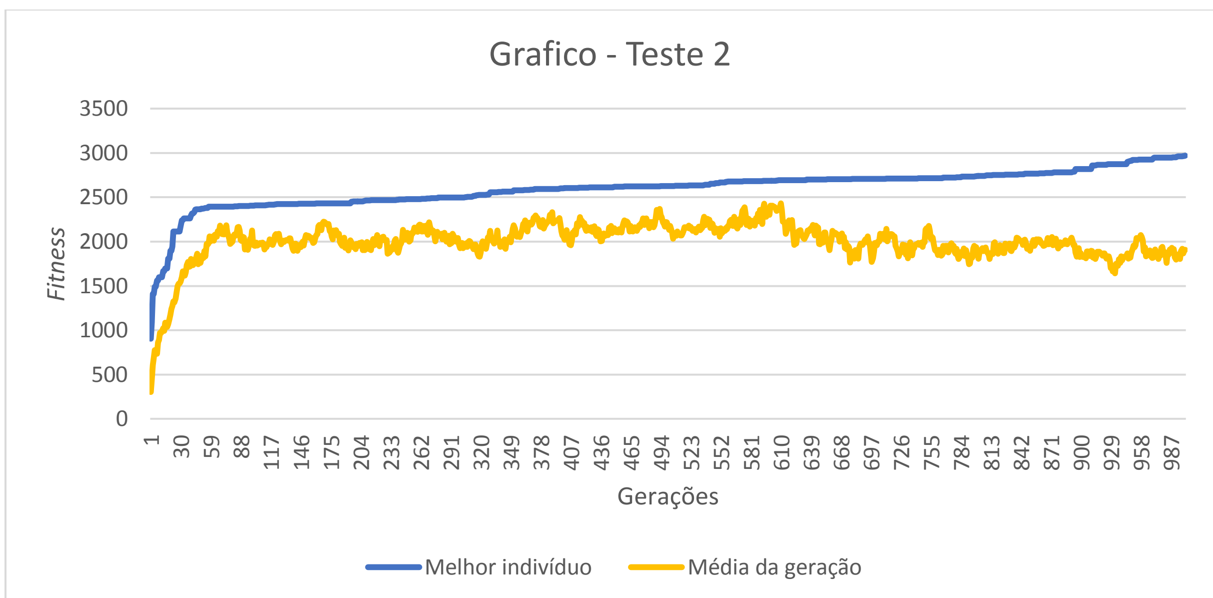

Gráfico 2 Representação da evolução do Teste 2

Teste 3 – Configuração:

- População inicial: 500 indivíduos
- Quantidade de pais selecionados: 30
- Quantidade de indivíduos criados pelos pais: 200
- Quantidade de indivíduos mutantes: 40

Resultado Teste 3

Tempo de execução: 149 minutos e 2 segundos

Chegou ao fim da fase? NÃO

No teste 3, representado pelo Gráfico 3, também executou as 1000 gerações, evoluiu melhor que o Teste 2 chegando a quase 5000 pontos de *fitness*, mas também não chegou ao fim da fase. Melhorou consideravelmente a distância entre o melhor *fitness* e a média da geração, mas demorou mais que o dobro do tempo para executar as 1000 gerações.

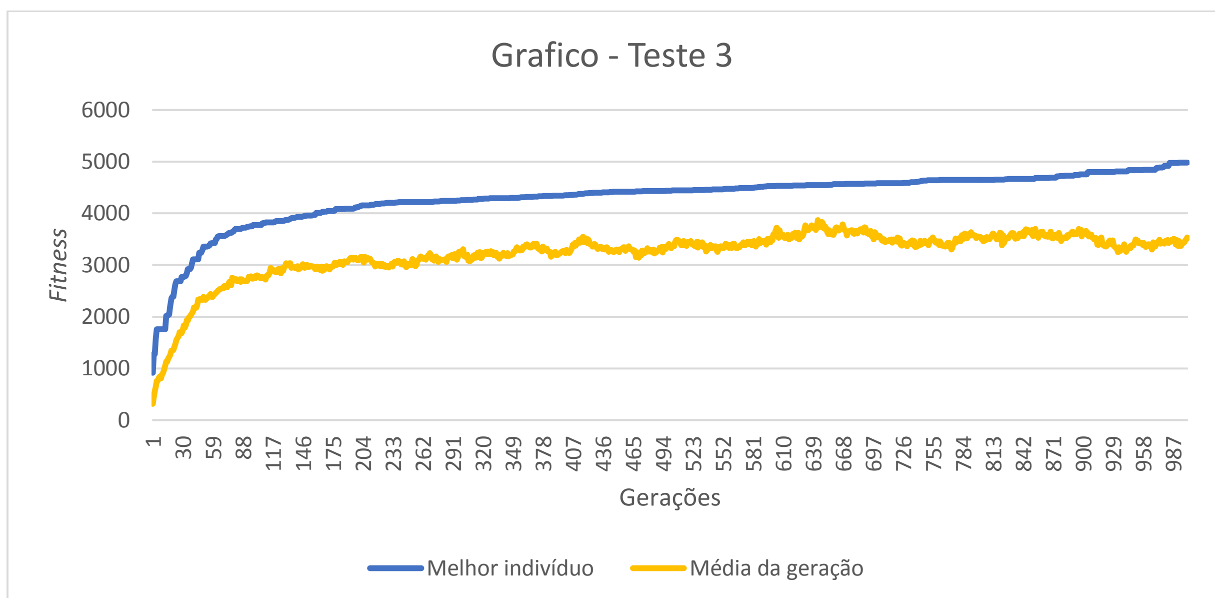

Gráfico 3 Representação da evolução do Teste 3

Teste 4 – Configuração:

- População inicial: 600 indivíduos
- Quantidade de pais selecionados: 70
- Quantidade de indivíduos criados pelos pais: 600
- Quantidade de indivíduos mutantes: 100

Resultado Teste 4

Tempo de execução: 33 minutos e 22 segundos

Chegou ao fim da fase? SIM

Representado pelo Gráfico 4, com essa configuração conseguimos fazer o algoritmo evoluir até que o Mario complete a fase, foram necessários 133 gerações e pouco mais de 33 minutos de execução para o algoritmo evoluir até encontrar a solução.

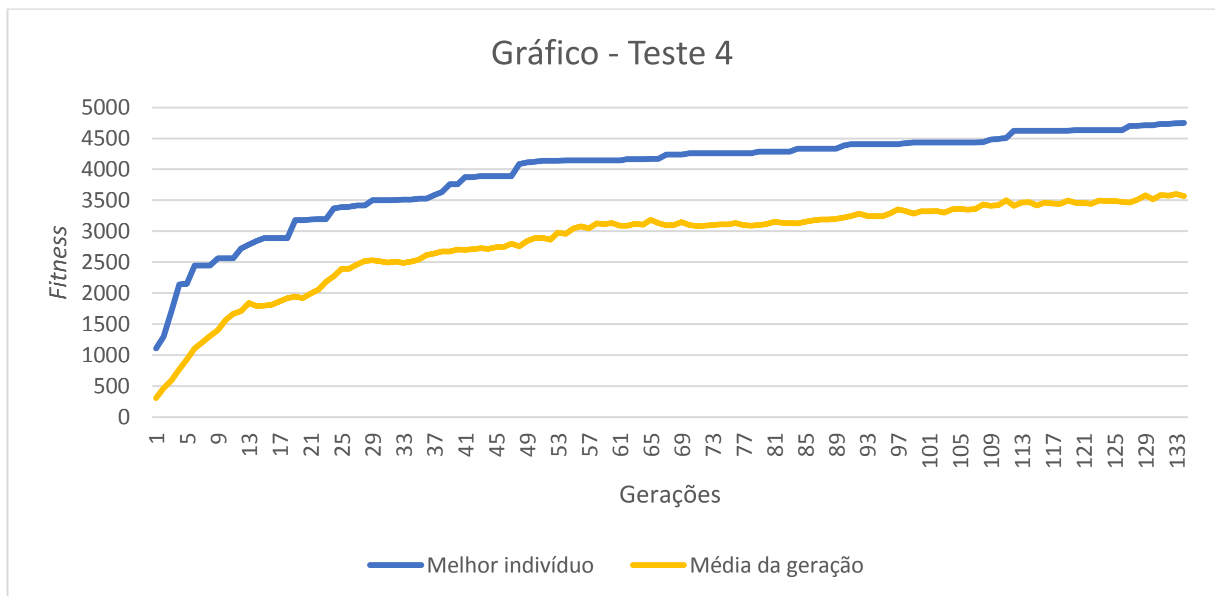

Gráfico 4 Representação da evolução do Teste 4- Configuração que completou a fase na 133º geração

Teste 5 – Configuração:

- População inicial: 600 indivíduos
- Quantidade de pais selecionados: 70
- Quantidade de indivíduos criados pelos pais: 600
- Quantidade de indivíduos mutantes: 100

Resultado Teste 5

Tempo de execução: 1.487 minutos e 59 segundos

Chegou ao fim da fase? SIM

O Teste 5 foi executado com as mesmas configurações do Teste 4, com o objetivo de avaliar a execução por mais gerações, deixamos executado por mais de 24 horas. Observamos no Gráfico 5 que o algoritmo continuou evoluindo bem até 350º geração, após isso a evolução foi mínima.

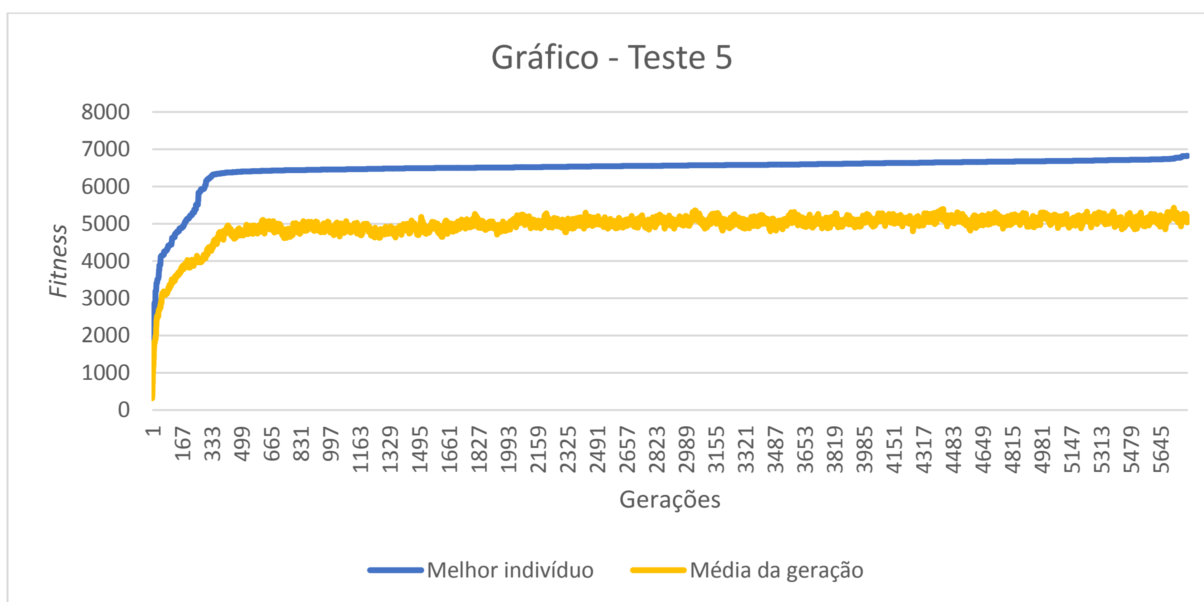

Gráfico 5 Representação da evolução do Teste 5

Após a execução dos 5 testes, percebeu-se que quanto maior o número de indivíduos que o algoritmo trabalha, melhor a população evolui a cada geração. Mas traz de desvantagem quanto maior a população maior o tempo de execução.

O algoritmo puramente genético se mostrou capaz de resolver o problema, mas cabe maiores estudos sobre como melhorar a função de avaliação, maiores testes com diferentes tamanhos de população, e a relação entre o tamanho da população e a quantidade de pais selecionados para *crossover*.

Outro ponto que pode melhorar o AG, é forçar que os cromossomos possuam somente movimentos em que levem o personagem Mario para direita, isso pode ocorrer na criação da população e nos operadores que geram a nova população, o *crossover* e a mutação. Mas deve-se tomar cuidado quanto a diversidade da população durante a evolução.

6. CONCLUSÃO

O objetivo de projetar e desenvolver um agente autônomo através dos algoritmo genético puro, para conseguir completar um nível do jogo Super Mario Bros, foi alcançada em sua totalidade. Para alcançar o objetivo foi necessário o estudo sobre o funcionamento dos algoritmos genéticos e suas diferentes formas de executar o método da seleção e o método de *crossover*. Após esse estudo modelamos o algoritmo genético ao Mario, isso incluiu definir como foi representado o cromossomo, modo de iniciar a população e a escolha dos métodos de seleção e *crossover*, e encontrar valores para configurar o AG. Esses valores foram encontrados através dos testes realizados.

Após a e dos testes, podemos concluir que o algoritmo genético desenvolvido foi capaz de reproduzir os resultados esperados.

6. REFERÊNCIAS

BARBOSA, C. J. **Estudo de Sistema Especialista Aplicado a Jogos**. Jaguariúna. Trabalho de conclusão de curso de graduação (Ciência da Computação) - Faculdade de Jaguariúna. 2007.

BOJARSKI, S; CONGDON, C B. Realm: **A rule-based evolutionary computation agent that learns to play mario.** In: Computational Intelligence and Games (CIG), 2010 IEEE Symposium on. IEEE, 2010. p. 83-90.

CLAESSENS, R. Mario AI - Gameplay Track Developing a Mario Agent. Não Publicado. 2012.

CUNHA, L. S.; GIRAFFA, L. M. M. **Um estudo sobre o uso de agentes em jogos computadorizados interativos.** Porto Alegre: Campus Global. Relatório Técnico Nº 017/2001, n. 017. 2001.

HOLLAND, J. **Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence**. U Michigan Press, 1975.

KARLSSON, B. F. F. **Um middleware de inteligência artificial para jogos digitais.** Dissertação de Mestrado, 2006. Departamento de informática. Pontifícia Universidade Católica do Rio de Janeiro. Rio de Janeiro. 2006

LACERDA, E. G. M.; CARVALHO, A. C. P. L. F. **Introdução aos algoritmos genéticos**. In: Galvão, C. O.; Valença, M. J. S. 1999 Sistemas inteligentes. Porto Alegre: Editora da Universidade da UFRGS – ABRH. 1999

LIMA, André Bezerra; AGUIAR, D. O.; DE SOUZA SARKIS, D. **Computação Evolutiva Aplicada A Jogos Eletrônicos**. 2012

LINDEN, R. **Algoritmos Genéticos (2a edição)**. [s.l.] BRASPORT, 2006.

MITCHELL, M. **An Introduction to Genetic Algorithms**. [s.l.] Bradford Books, 1998.

NORVIG, P.; RUSSEL, S. **Inteligência Artificial, 3ª Edição**. [s.l.] Elsevier Brasil, 2014.

PEREZ, Diego et al. **Evolving behaviour trees for the Mario AI competition using grammatical evolution**. In: Applications of evolutionary computation. Springer Berlin

Heidelberg, 2011. p. 123-132

PERSSON, M. Infinite Mario Bros. 2008.

SCHWAB, B. **Ai Game Engine Programming (Game Development Series)**. Charles River Media, Inc., Rockland, MA, USA. 2009

TOGELIUS, J et al. **The mario ai championship 2009-2012**. AI Magazine, v. 34, n. 3, p. 89-92, 2013.

TOGELIUS, J; KARAKOVSKIY, S; BAUMGARTEN, R. **The 2009 mario ai competition.** In: Evolutionary Computation (CEC), 2010 IEEE Congress on. IEEE, 2010. p. 1-8.