Algoritmos Genéticos (Junho 2019)

Jean Bertrand Paixão da Silva, *Estudante, UFRR*, Paulo Fábio dos Santos Ramos, *Estudante, UFRR*

***Abstract*—It will be done a study on genetic algorithms, addressing how they work and making practical applications for solving some problems found in the computing, such as the traveling salesman, and also the development of an AI for the game tetris.**

***Resumo*—Será feito um estudo sobre algoritmos genéticos, abordando a forma como eles trabalham e realizando aplicações práticas para resolução de alguns problemas encontrados na computação, como o caixeiro viajante e também será feito o desenvolvimento de uma IA para o jogo tetris.**

I. INTRODUÇÃO

O objetivo deste artigo é abordar sobre algoritmos genéticos, apresentando os conceitos básicos de como são implementados e demostrando como podem ser utilizados na pratica. Além disso, será feito uma análise e descrição sobre o artigo - Simulating nature's methods of evolving the best design solution. Cezary Janikow e Daniel Clair. IEEE. 1995; e também sobre o projeto Dinossauro da Google - https://www.youtube.com/watch?v=P7XHzqZjXQs; visando explicar em detalhes todas as fases do projeto.

Será feito também uma pequena introdução as classes de problemas que são encontrados na computação atualmente.

II. PROBLEMAS NP

Fazendo a análise da complexidade de algoritmos, conseguimos identificar diferentes classes de problemas, sendo elas P, NP, NP-difícil e NP-completo. A classe P de problemas, consiste naqueles que conseguem ser resolvidos em tempo polinomial, dado uma instancia do problema, ele sempre encontrara uma solução após determinado tempo. Problemas NP (nondeterministic polynomial) são aqueles que, dada uma entrada para o problema, é possível verificar se aquela entrada é faz parte de uma solução do problema. Ou seja, consiste na classe dos problemas de decisão, pois somente é necessário dizer se a solução proposta é válida ou não, podemos falar que $P \subset NP$.

Um problema é NP-Completo se ele for um problema NP-Difícil e também se ele for da classe NP, somente problemas de decisão podem ser classificados como NP-Completos. O conhecimento dos problemas NP-Completos se deu graças a Steven Cook, que na década de 70 conseguiu provar a existência do primeiro algoritmo dessa classe, conhecido como SAT (Problema de satisfatibilidade booleana), e a partir dele, se tornou provar a existência de outros problemas NP-Completos a partir da redução polinomial, pois dado um problema X e um problema Y tal que Y é NP-Completo, se provarmos que X é tão difícil quanto Y, X também é NP-Completo, isso consiste em dizer também que, se for possível resolver um problema NP-completo com um algoritmo determinístico em tempo polinomial, então todos os problemas da classe NP também podem ser resolvidos em tempo polinomial o que tornaria valida a questão "P = NP?", porém isso ainda não é possível.

III. PROBLEMAS NP-DIFÍCIL

Problemas NP-Difíceis são computacionalmente difíceis de se resolver, e até então não existe nenhum algoritmo que é capaz de resolver tais problemas em tempo polinomial. Um problema H é NP-difícil se existe um problema $B \in$ NP-completo, que pode ser transformado em H em tempo polinomial, ou seja, H é NP-Difícil caso seja uma variação de B, porém H não se encontra na classe NP.

IV. GENETIC ALGORITHMS – SIMULATING NATURE'S METHODS OF EVOLVING THE BEST DESIGN SOLUTION

Neste artigo o autor explica os conceitos básicos que constituem um algoritmo genético iremos destacar nesta descrição os pontos que mais chamaram a nossa atenção e que serviu de base para que pudéssemos concluir o trabalho final da disciplina Analise de algoritmo.

Gradualmente tem se resolvido problemas computacionais com agentes dinâmicos que interagem entre si por operações isoladas. Além disso alguns métodos vêm de forma natural onde organismos cooperam para o desenvolvimento de recursos. Isso para que seja desenvolvido um algoritmo com simulação de processos naturais. O Algoritmo Genético (GA) representa um dos melhores que tem uma aproximação correta.

Após essa introdução ele explica como funciona o algoritmo genético, que usa métodos para simular um processo natural, sendo esses métodos: seleção, informação de interação, mutação aleatória, e população dinâmica. Primeiro o Algoritmo Genético é melhor aplicado para parâmetros de otimizações numéricas, para facilitar o mapeamento do problema em forma de representar o mesmo em um determinado espaço.

A. GA's at a glance (A.G. de relance)

O algoritmo genético opera com uma simulação usando "agentes" individuais organizados em uma população onde lutam entre si por sobrevivência e cooperam para obter uma melhor adaptação. Os agentes são chamados de cromossomos. A estrutura de um cromossomo é composta

por genes. A maneira que definimos esses genes é extremamente importante pois é o que vai determinar se um cromossomo é uma potencial solução para o problema.

O algoritmo genético para achar a melhor solução usa seleção também conhecida como competição, onde cada cromossomo é avaliado individualmente e se os parâmetros obtidos por seus genes forem os melhores para solucionar o problema então esse cromossomo é selecionado e junto com outro que também satisfaça o problema é feito um cruzamento onde será gerado um cromossomo mais poderoso e robusto com genes desses dois cromossomos assim inicializando uma nova população.

A população inicial é gerada de forma aleatória, porem depois de selecionar os melhores cromossomos e fazer o cruzamento entre eles e adicionar uma mutação, onde é alterado alguns genes desses cromossomos de forma aleatória, é reproduzida uma nova população onde essa interação vai se repetir por diversas vezes onde o fator de pausa é definido pelo programador onde pode ser um tempo limite ou definido como uma quantidade de interações.

B. A theorical and intuitive look (uma visão teórica e intuitiva)

O algoritmo genético não funciona de forma aleatória. Ele explora regularidades no cromossomo representado. Na verdade, o cromossomo não é realmente individual, mas representa diferentes espécies. Dois cromossomos diferentes devem ter uma adaptação similar de espécies similares. Esses dois mesmos algoritmos têm diferentes valores de diferentes genes que são melhores adaptados. Para explorar esses cromossomos um diagrama é usado, um diagrama de modelo de similaridade fixado para alguns, mas alelos arbitrários para outros.

Infelizmente, o diagrama não consegue processar explicitamente porque ele não providencia um fenótipo completo necessário para avaliação. Em vez de um processo completo do algoritmo genético, para problemas práticos, todos os possíveis cromossomos não conseguem ser processados. Assim a informação sobre um cromossomo individual é generalizada para conclusões sobre um diagrama implícito.

A seleção por "pressão" causa uma pesquisa para processar por trabalho com um incremento da representação do cromossomo da média do diagrama. O processo continua por ter mais e mais diagramas específicos representando na população.

A seleção interativa termina quando a representação do diagrama converte para um único e especifico diagrama, ou seja, um cromossomo fixo, nenhum diagrama consegue ser preenchido porque não é representado pela população inicial. Para estender a pesquisa para outro diagrama, a operação de reprodução é utilizada. Reprodução causa exploração de novos diagramas como uma geração de novas instancias do presente diagrama.

Algoritmo genético trabalha por processamento implícito de diagramas por significar cromossomos explícitos, e o número de diagramas tem cromossomos representados na população de tamanho fixado de forma exponencial, algoritmos genéticos são ditos para exibir um paralelismo implícito. Além disso o algoritmo genético exibi paralelismo explicito, onde é processado em paralelo. Essa propriedade permite que o Algoritmo Genético utilize uma tecnologia rápida e avançada de processamento paralelo.

C. Applications(aplicações)

Algoritmos genéticos são aplicados para problemas que não conseguem ser resolvidos por outros que tem um custo menor. O algoritmo itself é uma simulação que não providencia respostas em tempo real. Além disso no geral é o único que consegue encontra uma "boa" solução, de aproximação de soluções.

No geral, uma aplicação de algoritmo genético requer:

1) Um claro entendimento do problema e seus objetivos.
2) Um algoritmo genético com:
   a) Cromossomo representado com semânticas definidas,
   b) Função de avaliação utilizando representação semântica e um mecanismo de seleção preciso, favorecendo a melhor solução,
   c) Uma população aleatória ou outra gerada representando os cromossomos,
   d) Operadores de reprodução, com algum mecanismo baseado em probabilidades estatísticas.

Um problema que melhor consegue ser expressado em termos adequados para algoritmos genéticos otimizados são:

1) Pesquisa por uma estrutura topológica.
2) Parâmetros numéricos otimizados.
3) Combinações destes dois.

Problemas práticos são melhor representados em uma combinação topológica e processamentos numéricos. Um exemplo é o problema de concepção e afinação de uma rede neural para feed-foward propagação. Aqui, o que buscamos é encontrar a solução que representa a melhor rede estruturada. Outro bom exemplo é o problema de otimizar regras difusas para controle ou classificação. Onde é necessário para processar "nível de pesos" das regras com algum componente numérico tal como uma regra "pesada".

V. DINOSSAURO GOOGLE

A ideia do projeto foi desenvolver uma inteligência artificial capaz de conseguir jogar de forma autônoma o joguinho do dinossauro da Google, fazendo ele desviar dos obstáculos que surgem no caminho e tentando alcançar a maior pontuação possível. Portanto é necessário que ele consiga enxergar o que estiver na frente e qual a distância que o objeto se encontra para que seja realizado o pulo, logo foram utilizados 3 sensores como entrada de dados para o sistema, um para medir a distância que o objeto se encontra, outro para medir o tamanho e outro para a velocidade, os 3 trabalhando em conjunto afim de determinar o melhor momento para a realização do salto, e além dos sensores, ele também conta com dois atuadores que no caso são as teclas para cima e para baixo do teclado para realizar as ações dentro do jogo

Para que uma máquina seja capaz de aprender de forma eficaz é necessário que se fuja da programação padrão, aquelas em que são executadas várias sequencias logicas seguindo um padrão predefinido pois isso é uma tarefa extremamente complicada, então é necessário abandonar essa regra e pensar

de forma mais abrangente, ou seja, em vez de pensarmos que um determinado dado possa assumir os valores de 0 ou 1, o que seria a forma discreta, podemos pensar que ele pode ser qualquer valor entre 0 e 1, adotando assim uma forma mais analógica, e para isso ser possível, em vez de ser feito uma vasta sequência de ifs e elses, podemos adotar a programação por meio de funções fugindo da forma mais linear, como por exemplo f(x) = g(A + Bx), onde x será os dados de entrada para a função, e A e B terão pesos que influenciaram diretamente no resultado final, em uma rede neural, podemos adotar vários pesos para A e B afim de se obter o melhor resultado, cada um desses pesos pode ser representado como um neurônio dentro da rede neural e a junção de todos eles garantem bons resultados a cada iteração. No caso do dinossauro da Google, as saídas a serem mapeadas são as setas, se o valor calculado for acima de 0.55 ele pressiona a tecla para cima e se for abaixo de 0.45 ele pressiona para baixo, caso esteja entre esses dois valores ele não faz nada.

Em uma rede neural, é possível sabermos exatamente qual o valor que está contido em cada neurônio, e com isso podemos pegar todos esses pesos e colocarmos eles de forma linear, como se fosse um DNA, que é uma sequência de informações sobre os seres vivos, no caso de uma máquina, podemos chamar essa abordagem de algoritmo genético, e cada um dos pesos da rede neural traz uma informação sobre como as entradas devem se comportar. Para dar início a aprendizagem em si, é necessário primeiro a criação de vários "genomas" que são compostos pelos pesos da rede neural, e a princípios esses pesos possuem um valor aleatório, à medida que o algoritmo for executado, será feita a escolha dos dois melhores genomas daquela geração e os demais serão descartados, o que pode ser chamado de "seleção natural" ou no caso, artificial, logo após essa escolha é feito um cross over, que consiste em pegar várias partes de cada um e combinar eles de forma que o novo genoma seja uma junção dos anteriores e além disso, para evitar que fique preso sempre nos mesmos resultados e aplicado uma função de modo aleatório em vários pesos do novo genoma, para que surjam algumas mudanças no geral, isso é chamado de mutação, obtendo assim um indivíduo completamente novo em relação aos pais dele, e são gerados vários dessa forma, logo em seguida esses novos genomas são colocados junto dos anteriores e então é feita uma nova análise para se obter os melhores genomas dessa geração e então é feito os mesmo passos até encontrar os melhores dos melhores. Para o dinossauro foram necessárias 120 gerações até ele se tornar um "ninja" e ainda é possível que se torne melhor.

VI. ALGORITMO GENETICO PARA O TETRIS

Tetris é um jogo mundialmente conhecido e consiste em empilhar blocos de diferentes formatos afim de completar as linhas horizontais da tela, sempre que uma linha se forma, a mesma se desintegra e o jogador ganha pontos, quando a pilha chega ao topo da tela o jogo acaba.

Com base nisso e seguindo um tutorial em java feito por Fernando Tenorio, foi possível desenvolver um algoritmo genético capaz de jogar o tetris de maneira autônoma, a seguir iremos explicar as classes que é utilizada o algoritmo genético.

A. Tetris agent

Essa é a classe base de nossa AI, e representa um jogador de Tetris. Três variáveis de instância são definidas: genes, clearedRowsArray e fitness. A habilidade de cada agente é expressa através de seus genes, onde cada slot da array genes corresponde ao peso atribuído a determinada característica da grade.

A array genes corresponde ao peso de 7 características distintas: número de linhas feitas, altura máxima da pilha e número de buracos não conectados, número de buracos conectados, número de blocos acima de buracos, número de poços e nivelação da grade.

Durante a execução da GA, cada agente jogará uma sequência de n games aleatórios, onde a sequência de peças em cada um dos n games será fixada semeando-se o gerador numérico. Em outras palavras, todos os agentes enfrentarão os mesmos jogos, o que nos permite balancear o efeito de um agente ruim receber uma sequência boa de peças, ou um agente bom receber uma sequência ruim de peças. A variável fitness quantifica as definições acima, e será calculada como a média de linhas removidas durante os n games.

Finalmente, temos a variável clearedRowsArray, que irá armazenar o número de linhas removidas em cada jogo executado em um a Thread separada. O problema em utilizar uma ArrayList padrão surge ao invocar agent.clearedRowsArray.add(linhas): caso duas Threads se esbarrem nesse método, uma ConcurrentModificationException seria lançada, terminando o programa.

Os métodos definidos são simples: eval recebe uma array com a configuração da grade, e utiliza os métodos estáticos da classe BoardEvaluator para avaliar a posição. A interface Comparable é implementada, nos permitindo ordenar agentes facilmente. O método estático randomAgent simplesmente cria um novo agente e seta seus genes com valores randômicos entre 0 e 1. Por fim, cloneAgent retorna um novo objeto com os mesmos genes do agente que invoca o método.

B. Classe BoardEvaluator

Esta classe define métodos estáticos, utilizados pelo agente para avaliar determinada configuração da grade. Os métodos clearedRows, pileHeight e countSingleHoles retornam o número de linhas removidas, a altura máxima, e o número de buracos, respectivamente.

O método countConnectedHoles retorna o número de buracos conectados, ou seja, buracos adjacentes são contados apenas uma vez. O método blocksAboveHoles retorna a quantidade de blocos que se encontram acima de buracos.

O método countWells retorna o número de poços, ou seja, regiões onde somente a peça do tipo I pode ser encaixada sem deixar buracos. Por fim, bumpiness calcula a nivelação da grade, observando as diferenças de altura entre colunas adjacentes.

C. Aplicando no algoritmo genético

Partindo de uma população randômica de k agentes, cada um será testado em uma série de n games aleatórios. A cada

geração, os agentes de maior desempenho terão uma probabilidade maior de serem selecionados para reprodução, o que dará origem aos membros da próxima geração, e assim sucessivamente. O algoritmo nos retornará uma lista com o histórico de todas as gerações.

D. Crossover

Para efetuar a troca de material genético é adotado uma técnica mista, combinando os genes dos pais em uma nova variável. O método blendCrossover da classe GeneticAlgorithm, faz exatamente isso. Escolhendo um ponto de corte aleatório, e a partir deste ponto, combinamos os genes correspondentes dos pais em um novo valor, sendo os genes dos outros filhos calculados da mesma maneira, porem utilizando 1-beta no lugar de beta.

E. Mutação

O método mutateNoise apenas percorre os genes, e introduz ou não, uma pequena variação noise. A probabilidade de um gene sofrer mutação, probMutate, é um dos parâmetros mais importantes da GA, e deve ser ajustado para cada problema.

F. Seleção

Agentes com melhor desempenho terão mais chances de serem selecionados para reprodução e passar seus genes adiante. O método doSelection usa roulette wheel selection para selecionar e retornar um agente.

G. GeneticAlgorithm

O método runGA executa o algoritmo, e retorna a ArrayList hist, onde cada item representa uma geração de agentes. Os parâmetros popSize, maxGens e probMutate representam respectivamente o número de agentes em cada geração, o número máximo de gerações e a probabilidade de ocorrer mutação em um gene. Os parâmetros gamesPerAgent e eliteRate representam respectivamente o número de jogos por agente, e a taxa de elitismo utilizada.

No loop principal começamos sorteando um inteiro que servirá como ponto de partida para semear o gerador aleatório. Em seguida, para cada agente, iniciamos um game passando como argumento o número de linhas e colunas, o tamanho do bloco, o valor para semear o gerador, e a variável doneSignal.

A variável doneSignal representa um objeto da classe CountDownLatch, inicializado com o número de Threads que desejamos esperar terminarem. Agora, ao invocar doneSignal.await(), podemos ter certeza que o programa só continuará a execução após todos os games/threads terem finalizado.

Em seguida, calculamos a média de linhas removidas por cada agente e atualizamos fitness e bestAgent. Seguindo com o código, alocamos a array que vai acomodar a nova população e invocamos sort na população atual. Agora os agentes estão ordenados de acordo com o desempenho quantificado em fitness (lembre-se que a classe TetrisAgent implementa a interface Comparable), e prosseguimos selecionando a elite, que automaticamente fará parte da nova população.

Por fim, o restante da nova população é preenchido: selecionamos indivíduos da população aos pares, efetuamos o crossover e mutação, e adicionamos os filhos a nova população. Ao final do loop adicionamos a população ao histórico, fazemos a troca population = newPopulation e incrementamos a geração atual.

H. Executando a GA

Dependendo dos valores usados em popSize, maxGens e gamesPerAgent, o algoritmo levará mais ou menos tempo para finalizar. É importante destacar que quanto maior os valores melhores a chance teremos de encontrar agentes bons ou otimos para resolver o tetris. Para o nosso teste oficial utilizamos os seguintes valores:

- popSize = 20
- maxGens = 30
- probMutate = 0.2
- gamesPerAgent = 10
- eliteRate = 0.2

Testamos o algoritmo em um notebook Dell (intel® Core™i3-5005U CPU @ 2.00GHz 2.00 GHz, 4,00 GB, sistema operacional de 64 bits) demorou cerca de 45 min para finalizar o programa. Porém é perfeitamente compreensível já que quanto melhor são os agentes mais demorado é para finalizar a partida

I. Resultados

Após executar o algoritmo foi gerado a seguinte tabela informando qual o melhor agente da geração e seus respectivos genes:

Melhor da geracao 1: 136.6	
0.45646352 0.8725518 0.966203 0.92830855 0.25838697 0.8653083 0.24421686	
Melhor da geracao 2: 281.5	
0.5854303 0.8491267 0.8030672 0.6941027 0.38517493 0.12382394 0.22393262	
Melhor da geracao 3: 287.5	
0.852022 0.9861275 0.89964795 0.8466147 0.34519652 0.47321483 0.25400174	
Melhor da geracao 4: 342.4	
0.45646352 0.8725518 0.966203 0.4887564 0.30008823 0.34397277 0.0019875467	
Melhor da geracao 5: 720.7	
0.5854303 0.563129 0.8030672 0.68688893 0.40529558 0.30565736 0.16471903	

Tabela com agentes de gerações de 1 ao 5

Fazendo o gráfico de todas as gerações testada teremos o seguinte resultado:

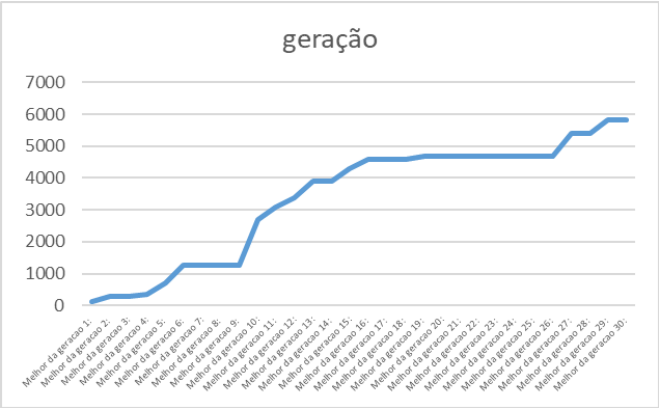

Gráfico representando o fitness de cada geração

Podemos observar que a partir da geração 22 o agente com melhor fitness se repete por outras gerações só voltando a aumentar novamente a partir da geração 25. Além disso podemos observar que o algoritmo genético realmente funciona selecionando sempre os melhores agentes para executar o cruzamento e a mutação para popular novamente uma nova geração.

VII. CAIXEIRO VIAJANTE

O problema do caixeiro viajante é determinado por, dado um número de cidades, qual o menor caminho a ser percorrido para visitar todas as cidades e retornar para a cidade de início, onde cada uma das cidades pode ser visitada uma única vez, um problema clássico de otimização combinatória e possui uma complexidade de crescimento de grau exponencial.

Com isso, a solução consiste em percorrer todas as rotas possíveis entre todas as cidades e selecionar a menor rota, tarefa que a primeira vista pode parecer fácil, porém se analisado com um pouco mais de calma perceberemos se tratar de um problema de classe NP-Completo, ou seja, podemos garantir que existe um caminho de menor custo, porem encontra-lo se torna um trabalho extremamente demorado, por exemplo, se usarmos um computador que seja capaz de realizar 1 bilhão de adições por segundo e se desejarmos encontrar uma solução através de força bruta para um mapa com 20 cidades, levaríamos cerca de 73 anos para o computador terminar de fazer todas as combinações possíveis, aumentando um pouco o número de cidades que se deseja visitar, é capaz do nosso universo colapsar e ainda assim o cálculo não ser terminado.

Vemos então que não é uma tarefa tão simples como parece, dito isso, foram propostas soluções de aproximação utilizando algoritmos genéticos, onde é possível encontrar uma ótima rota para visitar 100 cidades em menos de 1 minuto, dependendo de como for ajustado os parâmetros, isso se dá ao fato do algoritmo genético buscar sempre a melhor solução dado um determinado problema, no caso do TSP (Travelling salesman problem – Problema do caixeiro viajante), guardando as melhores rotas e fazendo uma serie de combinações entre elas afim de determinar novas rotas, melhores que as anteriores.

Para podermos solucionar o TSP através de algoritmos genéticos, é necessário seguir alguns passos, sendo eles:

1 – Criar um conjunto de rotas aleatórias, inicializando a população inicial de forma gulosa, ou seja, precisamos apenas montar um caminho entre todas as cidades, não importando a princípio o tamanho que cada rota irá possuir.

2 – Após a criação aleatória, será escolhida as duas melhores rotas da população para então ser realizado o cruzamento entre elas (Crossover), pois existem grandes chances dessas novas rotas serem melhores que os seus pais.

3 – Em alguns casos, as novas rotas irão sofrer o processo de mutação, isto acontece para que não exista duas rotas iguais dentro de uma população.

4 – Depois de todos os passos anteriores, as novas rotas são adicionadas na população ocupando assim o lugar das duas maiores rotas existentes.

5 – As roas filhas são criadas repetidamente até que se atinja um critério de parada, ou seja a melhor rota, mas corremos o risco de ficarmos travados em determinado estado sem nunca atingirmos a rota ideal, por isso é comum delimitar um número máximo de gerações e então selecionar a melhor rota nesse conjunto solução.

A. Crossover

Para o problema em questão, o crossover consiste em escolher um ponto especifico nas nossas rotas selecionadas e então fazer a combinação delas, tomando cuidado para que uma mesma cidade não apareça duas vezes nas novas rotas, caso isso ocorra, essa nova rota deve ser descartada

B. Mutação

O processo de mutação é interessante pois como só podemos trabalhar exatamente com o número de cidades correspondente a entrada de forma que elas não se repitam, nessa parte é feita somente uma mudança de posição entre duas cidades que são escolhidas aleatoriamente.

C. Avaliação/Aptidão

Nessa parte é feita a medição dos resultados, analisando o valor total do vetor solução

D. Seleção

Consiste em selecionar os melhores resultados, guardando esses valores, logo em seguida é feito novamente o passo a passo a partir do crossover com essas rotas sendo usadas como os pais para a nova geração.

VIII. COMPLEXIDADE

A complexidade média de um Algoritmo Genético não depende somente do tamanho da entrada n (por exemplo: no problema do caixeiro viajante, n é o número de cidades que devem ser percorridas, mas também dos parâmetros Numero de Gerações, Numero de indivíduos e ListaTaxa que pertencem à instancia do algoritmo. A complexidade do procedimento AG,C(AG), tem portanto esses 4 parâmetros, e analisado este procedimento conclui-se facilmente que:

Supondo que as linhas 2 a 4, do procedimento AG, são executadas nGerações vezes, tem-se a seguinte equação de complexidade:

$$C(AG)(n, nger, nind, listaTaxa) = C(inicializa)(n, nind) + \{nger.(C(condição - término)(n) + C(reprodução)(n, nindi, listaTaxa))\} + C(recupera)(n, nind)$$

Faz-se necessário, agora, calcular C(inicializa), C(condição-termino), C(reprodução) e C(recupera). Os procedimentos inicializa, condição-termino e recupera são procedimentos simples, cuja complexidade será determinada pela implementação destes. O procedimento reprodução é o elemento mais importante do Algoritmo Genético.

Na pratica, isso quer dizer que a complexidade para o algoritmo genético em relação ao caixeiro viajante pode ser definida em O(mn^2), onde m é definido pelo número de gerações e n é a quantidade de cidades a serem visitadas, enquanto que o método usando força bruta para percorrer cada caminho por vez possui complexidade igual a (n-1)!.

A. *Aplicando ao jogo tetris*

Temos nger = 30, nind = 20, listaTaxa = 0.2 e n = 10 sendo n o número de jogos por agente.

C inicializa possui complexidade O(n) pois terá que percorrer acrescentando valores aleatórios a todos os cromossomos.

C(condição de termino) = 1, pois para quando chegar na geração 30.

C(reprodução) = $\sum_1^n 1 = {n(n+1)}/{2}$

C(recupera) = $\sum_1^n 1 = {n(n+1)}/{2}$

$$= 30 * 20 + \left\{30 * \left(1 + \frac{n(n+1)}{2} * 10 * 20 * 0.2\right)\right\} + \frac{n(n+1)}{2} * 10 * 20$$

$$= 600 + \left\{30 * \left(1 + \frac{n^2+n}{2} * 40\right)\right\} + \frac{n^2+n}{2} * 200$$

$$= 600 + \{30 * (1 + 20n^2 + 40n)\} + 100n^2 + 100n$$

$$= 600 + 30 + 600n^2 + 1200n + 100n^2 + 100n$$

$$= 700n^2 + 1300n + 630$$

Logo, a complexidade é O(n^2)

IX. CONCLUSÃO

Podemos concluir que os algoritmos genéticos são ótimas opções para se resolver diversos tipos de problemas, sendo necessário que o programador tenha um pleno conhecimento sobre o problema a ser abordado para que então ele possa conseguir um código eficiente e por se tratarem de um método de evolução natural, retorna resultados bons com base no tempo de execução e a quantidade de gerações executadas onde quanto maior, melhores são os resultados porem o custo é bastante alto.

Os algoritmos genéticos são estocásticos e sua complexidade pode ser definida por:

O(O(Fitness) * (O(crossover) + O(mutation)))

Onde fitness é a função usada para verificar o critério de parada do algoritmo, em outras palavras, ela define o número a quantidade final de gerações que foram criadas, como esse valor pode nunca ser alcançado, outros fatores também determinam quando um algoritmo genético deve parar, mas isso se determina de forma isolada para cada tipo de problema a se analisado, crossover e mutação são as funções para geração de novos genes, ou seja, eles irão executar até que o critério de parada seja satisfeito.

REFERÊNCIAS

"Simulating nature's methods of evolving the best design solution". Cezary Janikow e Daniel Clair. IEEE. 1995;

"Inteligência artificial com dinossauro da Google". Ivan Seidel, disponível em < https://www.youtube.com/watch?v=P7XHzqZjXQs >. Acesso em 27 de junho 2019.

"Genetic Algorithm Tutorial - How to Code a Genetic Algorithm", Fullstack Academy, disponível em < https://www.youtube.com/watch?v=XP8R0yzAbdo>. Acesso em 27 de junho de 2019.

"Tetris e algoritmos genéticos em Java". Fernando Tenório, disponível em <http://bitsrandomicos.blogspot.com/2012/01/tetris-e-algoritmos-geneticos-em-java.html>. Acesso em 27 de junho de 2019.

"Problema do caixeiro viajante". J.F. Porto da Silveira, disponível em <http://www.mat.ufrgs.br/~portosil/caixeiro.html>. Acesso em 27 de junho de 2019.

"Resolvendo o problema do caixeiro viajante com Algoritmos Genéticos". Marcel Pinheiro Caraciolo, disponível em <http://aimotion.blogspot.com/2009/03/resolvendo-o-problema-do-caixeiro.html>. Acesso em 27 de junho de 2019.

"Um algoritmo genético para o TSP". N M S, disponível em <https://www.vivaolinux.com.br/script/Um-algoritmo-genetico-para-o-TSP-(Travel-Salesman-Problem)>. Acesso em 27 de junho de 2019.

"Analise formal da complexidade de Algoritmos Genéticos". Marilton Sanchotene de Aguiar, disponível em <>. Acesso em 27 de junho de 2019.