TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GABRIEL DE SOUSA MATSUMURA

RELATÓRIO DE ESTUDO:

ALGORITMOS GENÉTICOS

CARAGUATATUBA

2018

GABRIEL DE SOUSA MATSUMURA

RELATÓRIO DE ESTUDO:
ALGORITMOS GENÉTICOS

Relatório de estudo apresentado ao professor Dr. Valdivino Santiago Júnior, para avaliação de meu desempenho desde a nossa última conversa.

Orientador: Prof. Dr. Valdivino Santiago Júnior

CARAGUATATUBA

2018

SUMÁRIO

1 POSSÍVEIS CITAÇÕES

Seguem trechos selecionados do livro *An Introduction to Genetic Algorithms* de Melanie Mitchell (1999), que podem ser citados em futuros trabalhos:

Obs.: Há material de outra obra.

1.1 A Brief History of Evolutionary Computation, página 3:

Algoritmos Genéticos (“Genetic Algorthms”): “Os algoritmos genéticos (GAs) foram inventados por John Holland e colaboradores na década de 1960. O objetivo original de Holland era estudar o fenômeno da adaptação conforme ocorre na natureza e desenvolver maneiras de importar os mecanismos de adaptação natural em sistemas computacionais.

A obra *Adaptation in Natural and Artificial Systems* (Holland, 1975) apresentou o algoritmo genético como uma abstração da evolução biológica e forneceu uma estrutura teórica para a adaptação da evolução no GA. O método consiste em se deslocar uma população de cromossomos (por exemplo, vetores unidimensionais de bits) para uma nova população usando um tipo de 'seleção natural' inspirada nos operadores genéticos de cruzamento, mutação e inversão. Cada cromossomo é um conjunto de genes (por exemplo, bits), sendo cada gene uma instância de um alelo particular.”

Alelo: “É a representação (classe) do conjunto de possíveis características individuais que podem ser definidas pelo valor de um gene (objeto). Neste caso, o gene pode ser binário (ter dois estados) ou mais (como por exemplo a cor do cabelo ou dos olhos, que variam muito de um indivíduo para outro, este conjunto de variações é coberto pelo alelo).”

1.4 Search Spaces and Fitness Landscapes, página 6:

Espaço de Busca (“Search Space”): “É o ‘espaço de busca’ que contém uma coleção de soluções candidatas a uma solução desejada.”

Paisagem de aptidão (“Fitness Landscape”): “No contexto de genética populacional, a ‘paisagem de aptidão’ é a representação do espaço de todos os genótipos possíveis, junto às suas aptidões individuais (WRIGHT, 1931, apud op. cit.).”

Distância entre duas soluções (Hamming Distance): “É o número de locais em que os bits correspondentes diferem.”

1.5 Elements of Genetic Algorithms, página 7:

Componentes dos Algoritmos Genéticos: “Todos os Algoritmos Genéticos possuem: população de cromossomos, seleção de acordo com a aptidão, cruzamento para produção de novos descendentes, e mutações randômicas dos novos descentes.”

Página 8:

Pc e Pm: “Probabilidade de cruzamento/crossover (Pc) e mutação (Pm), respectivamente. Ambos as probabilidades com valores no intervalo de [0,1]”.

1.6 A Simple Genetic Algorithm, página 9:

Seleção Proporcional à Aptidão (“Fitness-Proportionate Selection”): “Método no qual o número de vezes que um indivíduo deve reproduzir é proporcional ao valor de sua aptidão dividida pela média de aptidão da população.”

1.7 Genetic Algorithms and Traditional Search Methods, página 10:

“Existem 3 tipos de busca: busca por dados armazenados, busca de caminhos para objetivos, e busca por soluções.”

Busca por dados armazenados (“Search for Stored Data”): “Neste caso o problema é encontrar dados armazenados em certa ordem na memória de um computador, a busca binária é um dos métodos para resolver de forma eficiente este problema.”

Busca de caminhos para objetivos (“Search for Paths to Goals”): “Neste tipo de busca o problema é encontrar um conjunto de ações que vai transitar de um dado estado inicial a um dado estado desejado, este tipo de busca é central para muitas abordagens de Inteligência Artificial.”

Página 11:

Busca por Soluções (“Search for Solutions”): “Neste tipo de busca a ideia é encontrar de forma eficiente uma solução dentre um amplo espaço de candidatos a solução.”

1.9 Two Brief Examples, página 19:

Reticulados (“Lattice”): “Reticulado como Relação de Ordem: Um reticulado é um conjunto parcialmente ordenado (CPO) no qual todo o par de elementos do conjunto possui simultaneamente soma e produto. Definição (baseada em CPO): Seja <P,R> uma relação de ordem parcial, então <P,R> é um reticulado se qualquer par de elementos de P possui simultaneamente menor limitante superior (soma) e maior limitante inferior (produto), ou seja: (∀ a ∈ P)(∀b ∈ P)(a*b ∈ P ^a+b ∈ P) (MENEZES, 2010).”

2.1 Evolving Computer Programs, página 28:

Programação Genética (“Genetic Programming”): “A ideia por trás da programação genética (GP) é evoluir programas que são difíceis de se escrever. Geralmente não se sabe de antemão quantas funções e terminais serão necessários para a escrita de um programa bem-sucedido.”

Funções e terminais (Functions and terminals): “Na programação genética de Koza (1992), expressões lógico-matemáticas são expressadas na estrutura sintática de árvore. Cada árvore consiste de funções (+, -, *, /, ^, v, dentre outros) e terminais (valores e variáveis).”

Página 30:

Programação Genética (“Genetic Programming”): “A técnica de GP tem sido bem-sucedida em evoluir corretamente programas para resolver um grande número de problemas simples (alguns nem tão simples) em diversos domínios, dentre eles a robótica, regressão simbólica, compressão de imagem, otimização, dentre outros.”

Sensores e atuadores (“Sensors and actions”): “Sensores aferem valores do ambiente (interno ou externo) de um sistema computacional. Podem ser entendidos como captadores de inputs. Já os atuadores tomam uma atitude (ativa ou passiva) frente ao processamento dos inputs. Atuadores podem ser entendidos como meios pelos quais os outputs se concretizam. Os sensores captam energia analógica, que devem se tornar digitais para que haja o seu processamento. Já os atuadores, recebem energia digital e a transformam em energia analógica.

Neste capítulo, Koza (1992) relaciona os conceitos funções e terminais com os conceitos de sensores e atuadores definidos por Nilsson (1989), neste caso os

terminais são um conjunto de sensores e as funções são um conjunto de atuadores. A união entre estes dois conjuntos forma o alfabeto utilizado na construção das árvores sintáticas.”

Página 34:

Cellular Automata (Autômatos Celulares): “A evolução natural cria sistemas nos quais ações de simples componentes com comunicação e informações limitadas geram um processamento global e coordenado de informações. Colônias de insetos, sistemas econômicos, o sistema imunológico e o cérebro foram citados como exemplos de sistemas nos quais esta forma de computação ocorre (Forrest 1990; Langton 1992). No entanto, ainda não se entende como esses sistemas naturais computam.”

Página 35:

Cellular Automata (Autômatos Celulares): “Autômatos celulares têm sido estudados extensivamente como objetos matemáticos, modelos de sistemas naturais e como arquiteturas para uma computação paralela confiável e rápida.”

2 ANOTAÇÕES

Como alguns capítulos foram bastante complexos para mim, como o *Hosts and Parasites: Using GAs to Evolve Sorting Networks* e 1.10 *How do Genetic Algorithms Work?*, exigindo diversas releituras para que eu entende-se ainda com dúvidas, em determinado momento, ao chegar ao *Evolving Cellular Automata*, após muitas releituras sem sucesso, comecei a anotar informações importantes para o entendimento. Essa estratégia tem dado certo e talvez essas anotações possam ser úteis mais tarde.

Evolving Cellular Automata, páginas 34-42:

- CA (Automato Celular): Reticulado de uma dimensão de N máquinas de 2 estados (“células binárias”);

- *Æ* (Regras do CA): Tabela de regras usada para atualizar a configuração do CA, simultaneamente em todas as células, conforme a variação do tempo;

- S_i: Estado local, o valor de uma única célula na posição i;

- Configuração: Padrão de estados locais de todo um lattice;

- r: Alcance de uma célula a células vizinhas. Exemplo: S_2 ^ r =1 => 0 ***1 0 1*** 0;

- $S_i = S_{(i+N)}$ (Condição de limite periódico): Significa que ao se iniciar o laço percorre-se de [S_i , $S_{(i+N)}$), ou seja, $S_{(i+N)}$ é a condição de parada;

- P_c = ½ (Performe a density classification-task): É a tarefa de classificar a densidade de bits nas células de um reticulado através da média aritmética;

- *Á*: É a densidade de bits em um reticulado em um determinado estado. $Á_c$ é a densidade crítica, sendo $Á_c = 0{,}5$. $Á_0$ é a densidade de um lattice no estado inicial (estado zero). A densidade é $(^{N-1}\sum_{i=0} S_{(i)})/N$. Se $Á_0 > Á_c$, então após M estados o reticulado deve ter $Á_M = 1$. Se $Á_0 < Á_c$, então após M estados o reticulado deverá ter $Á_M = 0$. Logo, $\{(Á_0 > Á_c <=> Á_M = 1) \wedge (Á_0 < Á_c <=> Á_M = 0) \mid 0 \leq M \leq N-1\}$. O objetivo é encontrar uma ou mais *Æ*(tabelas de regras) para satisfazer as condições da expressão acima.

- IC (Configuração Inicial): $Á_0 \neq$ IC, porque IC é a sequência ordenada de bits em um lattice inicial, enquanto que $Á_0$ é a densidade inicial que varia entre [0 , 1], ou seja, ICs diferentes podem ter o mesmo $Á_0$.

Packard's (1988) GA, página 37:

- Os cromossomos que serão evoluídos representam *Æ* (tabelas de regras) e consistem apenas dos bits de saída para cada possibilidade de avizinhamentos;

- O tamanho dos cromossomos é definido pela fórmula 2^{2r+1}, neste caso r = 3, logo o espaço de busca é de 2^{128};

- Os reticulados têm tamanho N = 149;

- A população inicial é de 100 cromossomos tendenciosamente randomizados;

- Para o cálculo da aptidão:

(i) Escolhe-se randomicamente 100 ICs (configurações iniciais) uniformemente distribuídas em um vetor bidimensional *ÁÎ*[0.0,1.0], sendo *ÁÎ*[0] um IC com $Á_0 < Á_c$ e *ÁÎ*[1] um IC com $Á_0 > Á_c$. Os bits são uniformemente distribuídos entre *ÁÎ*[0] e *ÁÎ*[1]. Neste momento é possível que *ÁÎ*[1] seja preenchido por NOT *ÁÎ*[0], caso não seja isto com certeza a randomização é trabalhada para que $ÁÎ[0] < Á_c < ÁÎ[1]$. Nenhuma IC terá densidade igual a $Á_c$ porque N é ímpar;

(ii) Testar as *Æ*s nas ICs até que se chegue a uma densidade fixa ou até 2*(N-1) iterações, pois na verdade cada ÁÎ tem duas ICs;

(iii) Determinar se o padrão final está correto – se a IC tiver $Á_0 < Á_c$ então $Á_M = 0$, se não $Á_M = 1$. A aptidão vai variar de [0, 100], pois cada *Æ* será testada 100 vezes, se uma *Æ* for bem-sucedida em uma IC, então terá 1 incrementado a sua aptidão, caso contrário incrementa-se 0;

- O algoritmo genético funciona como se segue:

(i) A cada geração um novo conjunto de ICs são gerados;

(ii) A função de aptidão é calculada para cada *Æ* na população;

(iii) A população é ordenada conforme sua aptidão;

(iv) As 20 *Æ* de maior rank (regras de elite) serão copiadas para a próxima geração sem modificação;

(v) As 80 regras para a próxima geração serão formadas pelos cruzamentos de ponto único entre pares de regras de elite escolhidas randomicamente. A mesma regra de elite pode cruzar diversas vezes. O descendente de cada cruzamento sofre mutação duas vezes. O algoritmo “corre” por 100 gerações;

- Como as *Æs* evoluídas por este GA são ineficientes ao tentar o Pc = ½ de ICs não tendenciosas, Mitchell et al (1994) fizeram melhorias testes sobre os resultados obtidos com a GA de Packard (1988). Inclusive, na época desta edição

Mitchell e col. Trabalhavam em melhorias no GA de Packard, utilizando o conceito de hospedeiros e parasitas para que os *Æ*s e as ICs coevoluíssem;

Mitchell and col.'s (1994) tests above Packard's GA, página 39:

- $P_N(\phi)$ (unbiased performance) – É a fração das classificações corretas produzidas por uma *Æ* após aproximadamente 2N iterações em cada um das 10000 ICs geradas com uma distribuição não tendenciosa sobre *Á*;

- Nesta etapa, ao que parece, as *Æ*s de maior aptidão, obtidas com a GA de Packard foram testadas em $P_N(\phi)$ para que houvesse uma ideia de seu desempenho frente á IC's não tendenciosamente geradas e de tamanho N maiores. Desta forma houveram 4 categorias de GA, fora as *Æmaj* (maioria das regras) que obteve zero de performance. Sendo as categorias *Æa, Æb, Æc, Æd*, houve uma performance crescente de *Æa* até *Æd.* Não ficou claro se todas estas foram geradas com o GA de Packard. O interessante é que, fora a *Æa* que é uma regra que opera expandindo blocos, as demais regras (*Æb, Æc* e *Æd*) operam baseadas em partículas do reticulado, sendo capazes de propagar sinais ao longo do tempo entre as partículas, com informações sobre as densidades locais;

- É extremamente difícil descobrir qual o algoritmo por trás das regras evoluídas pelo GA, sendo este um problema comum na computação evolutiva. Uma abordagem promissora é examinar os padrões espaçotemporais das *Æ*s operando sobre ICs para reconstruir o algoritmo da regra. O método de Crutchfield e Hanson (1993), chamado "Mecânica Computacional", trata do processo de reconstrução algorítmica em termos de "domínios regulares", "partículas" e "interações entre partículas", fazendo uma analogia aos conceitos da Teoria da Linguagem Formal;

- Domínios regulares são regiões espaçotemporais que consistem em palavras de uma mesma linguagem, sendo dados computacionalmente simples. Partículas são limites localizados entre domínios regulares que possuem como papel o transporte de informação. No momento em que duas partículas colidem ocorre o processamento das informações por elas carregadas. Partículas e suas interações formam uma linguagem de alto nível útil para descrever a computação realizada por sistemas como os autômatos celulares;

- A análise de partículas foi usada por Mitchell et al (1994) para compreender o processo de evolução de *Æ*s realizado pelo GA. As regras de maior aptidão evoluídas pelo GA, como a *Æd*, com um número maior de gerações sendo testadas

por mais ICs, chegaram a uma performance muito próxima à da regra de maior aptidão da época, a $Æ_{GKL}$ (Gacs−Kurdyumov−Levin) descoberta por Gacs et al (1978) para o estudo de computações confiáveis, sendo a principal diferença de performance entre a *Æd* e a $Æ_{GKL}$ devida as assimetrias presentes na *Æd*. A análise de partículas tornou possível determinar regras de aptidões maiores que a $Æ_{GKL}$;

- Com o estudo do funcionamento do GA está sendo possível compreender a quebra de simetrias da evolução natural, que alcança estratégias computacionais de nível subótimo.

3 DÚVIDAS

Seguem trechos que me causaram dúvidas ainda não resolvidas:

1.10 How do Genetic Algorithms Work?, página 22.

Estava compreendendo bem a noção de esquemas (blocos de construção), até chegar nas fórmulas matemáticas da página 22, as entendi parcialmente.

Chapter 1 – Thought Exercises, página 24.

3. How many possible sorting networks are there in the search space defined by Hillis's representation?

Eu até consegui entender boa parte dos métodos de Hillis apresentados no capítulo *Hosts and Parasites: Using GAs to Evolve Sorting Networks*, mas fiquei com dúvidas a respeito dessas redes de ordenação: um individuo para Hillis tem 15 pares de cromossomos, e cada cromossomo tem 32 bits. Um par de cromossomos já é o suficiente para representar números inteiros de [0,15], mas um par de cromossomos não é uma rede de ordenação? Apenas um indivíduo com 15 pares de cromossomos é uma rede de ordenação? Bem, se apenas um par de cromossomos for considerado uma rede de classificação isso significa que o espaço de busca é de 2^{64}. Bem creio que esse número não é muito desafiador, se um indivíduo for uma rede de ordenação, então creio que a resposta é $2^{15*64} = 2^{960}$. Aqui creio que já é um valor absurdo, mas não sei qual dos dois está correto e ainda pode ser que uma rede de ordenação não seja nem um indivíduo e nem um par de cromossomos.

4 THOUGHT EXERCISES

Seguem questões que consegui responder até o momento, no entanto elas não estão corrigidas porque no livro não há respostas:

Chapter 1: Genetic Algorithms: An Overview, página 23:

1. How many Prisoner's Dilemma strategies with a memory of three games are there that are behaviorally equivalent to TIT FOR TAT? What fraction is this of the total number of strategies with a memory of three games?

Resposta: Para começar, o total de estratégias possíveis com uma memória de 3 jogos é de 2^6 (64), enquanto que o total de estratégias possíveis com uma memória de um jogo são 2^2.

No caso do TIT FOR TAT com memória de 1 jogo, a estratégia é representada pela string CDCD, sendo “C” Cooperar e “D” Denunciar. Essa string possui 4 posições binárias, sendo que cada posição representa a “jogada” a ser realizada frente a memória do jogo anterior.

Considerando como bits definidos apenas os casos nos quais as três memórias de jogos anteriores são iguais, pois as variações do TIT FOR TAT com a memória de três jogos podem não ter uma ação imediata a jogada anterior, obtive 16 bits definidos dentro de uma sequência de 64 bits. A fração é 2^{-48}.

2. What is the total payoff after 10 games of TIT FOR TAT playing against

(a) a strategy that always defects; **Resposta:** 9 pontos.

(b) a strategy that always cooperates; **Resposta:** 30 pontos.

(c) ANTI−TIT−FOR−TAT, a strategy that starts out by defecting and always does the opposite of what its opponent did on the last move? **Resposta:** 19 pontos.

(d) What is the expected payoff of TIT FOR TAT against a strategy that makes random moves? **Resposta:** Entre 9 e 30 pontos.

(e) What are the total payoffs of each of these strategies in playing 10 games against TIT FOR TAT? (For the random strategy, what is its expected average payoff?) **Resposta:** Aways defects: 14 pontos; Aways cooperates: 30 pontos; ANTI−TIT−FOR−TAT: 24 pontos; Random: Uma média de 22 pontos.

5 COMPUTER EXERCISES

Seguem questões de programação que consegui resolver:

1. Implement a simple GA with fitness−proportionate selection, roulettewheel sampling, population size 100, single−point crossover rate pc = 0.7, and bitwise mutation rate pm = 0.001. Try it on the following fitness function: f(x) = number of ones in x, where x is a chromosome of length 20. Perform 20 runs, and measure the average generation at which the string of all ones is discovered. Perform the same experiment with crossover turned off (i.e., pc = 0). Do similar experiments, varying the mutation and crossover rates, to see how the variations affect the average time required for the GA to find the optimal string. If it turns out that mutation with crossover is better than mutation alone, why is that the case?

Resposta: Segui os passos de Souza (2015) para que eu pudesse ver pela primeira vez a lógica por trás da implementação dos algoritmos genéticos. O algoritmo desenvolvido por Souza (2015) é muito próximo ao requisitado por esta questão. Bastou para mim mudar da linguagem C++ para Java, adicionar alguns métodos como o *roundRobin()*, que foi útil para tornar a seleção natural proporcional as aptidões existentes em cada geração e modificações como a do método *fitness()*, no qual acrescentei um parâmetro booleano para alterar a função de fitness para a da questão 2. Segue o código:

2. Implement a simple GA with fitness−proportionate selection, roulettewheel sampling, population size 100, single−point crossover rate pc = 0.7, and bitwise mutation rate pm = 0.001. Try it on the fitness function f(x) = the integer represented by the binary number x, where x is a chromosome of length 20. Run the GA for 100 generations and plot the fitness of the best individual found at each generation as well as the average fitness of the population at each generation. How do these plots change as you vary the population size, the crossover rate, and the mutation rate? What if you use only mutation (i.e., pc = 0)?

3. Define ten schemas that are of particular interest for the fitness functions of computer exercises 1 and 2 (e.g., 1*⋯* and 0*⋯*). When running the GA as in computer exercises 1 and 2, record at each generation how many instances there are in the population of each of these schemas. How well do the data agree with the predictions of the Schema Theorem?

4. Compare the GA's performance on the fitness functions of computer exercises 1 and 2 with that of steepest−ascent hill climbing (defined above) and with that of another simple hill−climbing method, "random−mutation hill climbing" (Forrest and Mitchell 1993b):

1 - Start with a single randomly generated string. Calculate its fitness;

2 - Randomly mutate one locus of the current string;

3 - If the fitness of the mutated string is equal to or higher than the fitness of the original string, keep the mutated string. Otherwise keep the original string;

4 - Go to step 2.

Iterate this algorithm for 10,000 steps (fitness−function evaluations). This is equal to the number of fitness−function evaluations performed by the GA in computer exercise 2 (with population size 100 run for 100 generations). Plot the best fitness found so far at every 100 evaluation steps (equivalent to one GA generation), averaged over 10 runs. Compare this with a plot of the GA's best fitness found so far as a function of generation. Which algorithm finds higher−fitness chromosomes? Which algorithm finds them faster? Comparisons like these are important if claims are to be made that a GA is a more effective search algorithm than other stochastic methods on a given problem.

REFERÊNCIAS

SOUZA, M. C. de. Curso de C++ - Aula 89 - Algoritmos Genéticos - **Problema OneMax.** Youtube, 2015. Disponível em: <https://youtu.be/ZLQjr_8Epzs>. Acesso em: 25 de fevereiro de 2018.

MENEZES, P.B. **Matemática discreta para computação e informática.** Porto Alegre: Bookman, 2010.

MITCHELL, M. **An Introduction to Genetic Algorithms.** Cambridge: MIT, 1999. Disponível em: <http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf>. Acesso em: 25 de fevereiro de 2018.