

Trabalho Prático 3 de Inteligência Artificial – Problema do Passeio do Cavalo

Aluno: Diego Ascânio Santos

Professor: Flávio Vinícius Cruzeiro Martins

Introdução:

O problema do passeio do cavalo é um dos quebra cabeças mais tradicionais de todos os tempos e esse problema consiste em movimentar em 'L' um cavalo em um tabuleiro de xadrez (ou qualquer outro tabuleiro $n \times n$) de modo a visitar todas as casas do tabuleiro sem repetir nenhuma delas. O problema é extremamente antigo e tem-se conhecimento de que ele existe ao menos desde o século IX depois de Cristo.

O trabalho consiste em implementar um algoritmo genético para solucionar esse problema em tabuleiros $n \times n$ usando as técnicas aprendidas em sala de aula.

Algoritmo Genético:

Um algoritmo genético consiste em tratar possíveis soluções de um problema de otimização como indivíduos de uma população e submeter tais indivíduos a leis de evolução inspiradas na evolução darwiniana para que esses indivíduos se reproduzam, mutem e compitam entre si pela sobrevivência até uma geração limite. Pelas leis da evolução, os indivíduos mais adaptados ao meio (ou seja, os melhores indivíduos para aquele contexto) tendem a perseverar e repassar seus genes às próximas gerações, ou seja as próximas gerações conterão indivíduos melhores (mais aptos) que as anteriores. Tratando-se de soluções para problemas de otimização, a cada geração, as soluções que persistem são aquelas que se aproximam ou se igualam a solução ótima do problema de otimização.

Tecnologias adotadas:

Linguagem Python 2.7

A preferência pela linguagem python ocorre pela grande gama de recursos que a linguagem oferece para o desenvolvedor e pela proximidade da linguagem com o pseudo código o que facilita e muito o entendimento do código, tornando-se um código limpo e claro em relação ao que faz

Modelagem do Problema e Desenvolvimento do Algoritmo:

Em pesquisa a internet sobre resoluções do problema do passeio do cavalo com algoritmos genéticos, encontrou-se um excelente artigo vinculado ao departamento de informática e ciência da computação da Universidade Estadual do Rio de Janeiro (UERJ) de autoria dos senhores Fernando Tamberlini Alves e Paulo Eustáquio Duarte Pinto. O título do artigo é ***Aplicação de Algoritmos Genéticos ao Problema do Percorso do Cavalo*** e esse artigo serviu de base para a modelagem da solução de algoritmo genético implementada, usando das mesmas abordagens explicitadas no artigo para a construção de funções de avaliação, geração, seleção, cruzamento, mutação e renovação de indivíduos.

Durante a leitura do artigo, constatou-se a necessidade de implementar funções para auxiliar a construção do algoritmo genético, pois, sem tais funções, a construção do AG seria mais árdua. Essas funções encontram-se no arquivo ferramentas.py e elas realizam tarefas como converter uma casa (x, y) de um tabuleiro $n \times n$ para um numeral que representa a sua ordem no tabuleiro (e vice-versa), retornar

as adjacências de uma casa, retornar os movimentos legais do cavalo a partir de uma casa e retornar um dicionário de movimentos legais para o cavalo em todas as casas de um tabuleiro nxn.

No arquivo `genetico.py` implementou-se a classe `AlgoritmoGeneticoPasseioCavalo`, contendo os métodos e atributos necessários a execução do AG, assim como o próprio método de execução do AG.

O Construtor da classe AG recebe como parâmetros `n`, `tamanho_populacao`, `geracoes`, `probabilidade_mutacao` e uma função opcional de fitness. O parâmetro `n` no caso é a raiz quadrada do número de casas do tabuleiro (nxn)

O método de execução do AG seguiu como base o pseudo código proposto por Alves e Pinto e a sua implementação em python encontra-se abaixo:

```
def executar(self):
    """ para executar o AG """
    self.iniciar_populacao()
    self.avaliar_populacao()
    Otimo = False

    for g in range(self.geracoes):
        self.selecionar_populacao()
        self.reproduzir_populacao()
        self.mutar_populacao()
        self.avaliar_populacao()
        self.renovar_populacao()
        # teste de condicao para estao Otimo
        if self.fitness(self.populacao[0]) == (self.tamanho_individuo - 1):
            Otimo = True
            break

    return self.populacao[0], Otimo
```

Figura 1: Metodo de Execução do AG

O primeiro comando do método `executar`, chama o método `iniciar população`, que gera os indivíduos que serão incluídos na população do algoritmo genético.

Um indivíduo nesse algoritmo genético é uma tupla (tamanho nxn) de casas no tabuleiro representando uma sequência de movimentações (possível ou impossível) do cavalo.

As casas nessa tupla podem assumir dois formatos: coordenadas no tabuleiro (x, y) ou numérica (representando a ordem da casa no tabuleiro – 1 → (1,1), 2 → (1, 2) e assim por diante.).

```
>>> print individuo_numerico
(17, 6, 3, 10, 19, 22, 11, 2, 9, 20, 13, 24, 15, 4, 7, 16, 23, 12, 1, 8, 5, 14, 25,
18, 21)
>>> print individuo_coordenadas
((4, 2), (2, 1), (1, 3), (2, 5), (4, 4), (5, 2), (3, 1), (1, 2), (2, 4), (4, 5), (3,
3), (5, 4), (3, 5), (1, 4), (2, 2), (4, 1), (5, 3), (3, 2), (1, 1), (2, 3), (1, 5),
(3, 4), (5, 5), (4, 3), (5, 1))
```

Figura 2: Exemplo de individuos

Os indivíduos são gerados utilizando-se de 3 abordagens: totalmente aleatória, aleatória entre vizinhos disponíveis e baseada na regra de warnsdorff.

A abordagem totalmente aleatória consiste em realizar uma permutação aleatória das casas de um indivíduo, começando inicialmente na ordem de 1 → tamanho do indivíduo. A abordagem aleatória

entre vizinhos disponiveis consiste em escolher uma casa aleatoria no tabuleiro e a partir daí, visitar adjacencias, respeitando a regra de não ir para adjacencias já visitadas, até percorrer o tabuleiro inteiro ou entrar em um beco sem saída. Quando a segunda situação ocorre, deve-se escolher uma casa aleatória das casas ainda não visitadas no tabuleiro e continuar o processo, até que todas as casas tenham sido visitadas. Já a abordagem de warnsdorff consiste em escolher uma casa aleatória inicialmente no tabuleiro e se dirigir para casas legais (na movimentação do cavalo) que tenham o menor número de casas não visitadas. Essa é uma heurística muito boa e em grande parte dos casos, encontra a solução para o problema do passeio do cavalo e existem pouquíssimas situações onde o cavalo vai para um beco sem saída. Quando ocorre alguma dessas situações, seleciona-se uma casa aleatória das casas não visitadas no tabuleiro e continua-se o processo de warnsdorff, sempre respeitando a regra de mover-se para casas legais que tenham o menor número de casas não visitadas. Após todos os indivíduos da população terem sido obtidos, em iniciar população ocorre um embaralhamento da população para garantir uma maior heterogeneidade nas etapas posteriores.

Após a população ter sido iniciada, ela é avaliada e classificada no método avaliar_populacao() que preenche um dicionário de aptidoes para cada um dos indivíduos da população corrente. Essa aptidao é obtida pela função fitness, que pode ser uma a gosto do usuário ou a própria implementada no algoritmo genético, que retorna como aptidão de um indivíduo o tamanho da maior sequencia de movimentos válidos que tal indivíduo possui.

```
def __fitness(self, *args):
    """
    Uma funcao para calcular o quanto vale um determinado individuo da
    populacao. Esse valor pode ser atribuido com a maior quantidade de
    movimentos legais que esse individuo possui
    Entrada: individuo
    Saída: aptidao
    """

    individuo = args[0]

    aptidao = 0
    aptidao_local = 0
    i = 0
    j = 0

    movimentos_legais = deepcopy(self.movimentos_legais)

    while i < len(individuo) - 1:
        j = i
        while j < len(individuo) - 1 and individuo[j + 1] \
            in movimentos_legais[individuo[j]]:
            aptidao_local += 1
            if aptidao < aptidao_local:
                aptidao = aptidao_local
            j += 1
        i = j + 1
        aptidao_local = 0

    return aptidao
```

Figura 3: Função Fitness da classe AG

Depois da população ter sido avaliada e classificada, inicia-se a execução do algoritmo genético em si, onde ocorrerá o processo de seleção de indivíduos, reprodução, mutação, classificação e renovação da população.

O processo de seleção, que ocorre em selecionar indivíduos, consiste em percorrer cada membro da população e situá-lo em uma roleta (pois a seleção ocorre utilizando o método da roleta). Após isso, a roleta é girada **o tamanho da população** vezes para selecionar elementos que irão reproduzir. Pelo fato do método da roleta ser utilizado, existe a possibilidade de que um mesmo elemento seja selecionado mais de uma vez, podendo ocorrer inclusive a auto fertilização. Mas isso não é um problema, uma vez

que o método da roleta é democrático, permitindo selecionar indivíduos de todos os tipos para a reprodução.

```
def selecionar_populacao(self):
    """
    Metodo que seleciona os individuos de uma populacao
    pelo metodo da roleta
    """
    valor_total = 0
    roleta = {}
    individuos_selecionados = []

    for p in self.populacao:
        valor_total += self.valor_individuos[p]
        roleta[p] = valor_total

    for p in self.populacao:
        giro = random.randint(0, valor_total)
        for q in self.populacao:
            if giro <= roleta[q]:
                individuos_selecionados.append(q)
                break

    self.populacao = individuos_selecionados
```

Figura 4: Método de seleção de indivíduos para reprodução

A reprodução ocorre no método reproduzir_populacao utilizando-se a estratégia de reprodução por ponto duplo de corte, onde o corte irá ocorrer na maior sequência de movimentos válidos de um dos pais.

Os elementos são pareados de dois em dois em duas listas – pais e maes e seus descendentes são gerados pela estratégia supracitada. Após gerar todos os descendentes dos indivíduos pareados, executa-se o método corrigir_descendentes para corrigir os indivíduos que apresentaram casas repetidas. A correcao é bem simples, substituindo-se as ocorrências excedentes de casas por casas que tem zero ocorrências naquele indivíduo.

```
# para individuos de dois em dois
pais = []
maes = []
for i in range(0, self.tamanho_populacao, 2):
    pais.append(self.populacao[i])
    maes.append(self.populacao[i + 1])

# começa a gerar os descendentes
for i in range(0, self.tamanho_populacao/2):
    # determina sequencias de movimentos legais possiveis
    sequencias = [calcula_ponto_duplo_de_corte(pais[i]), \
                  calcula_ponto_duplo_de_corte(maes[i])]

    # pega o inicio e o final da maior sequencia
    inicio_sequencia, fim_sequencia = \
    max(sequencias, key=lambda x: x[1] - x[0])

    # primeiro descendente - pai, mae, pai
    descendente_1 = ()
    descendente_1 += tuple(pais[i][0:inicio_sequencia])
    descendente_1 += tuple(maes[i][inicio_sequencia:fim_sequencia])
    descendente_1 += tuple(pais[i][fim_sequencia:])

    # segundo descendente - mae, pai, mae
    descendente_2 = ()
    descendente_2 += tuple(maes[i][0:inicio_sequencia])
    descendente_2 += tuple(pais[i][inicio_sequencia:fim_sequencia])
    descendente_2 += tuple(maes[i][fim_sequencia:])

    self.descendentes.append(descendente_1)
    self.descendentes.append(descendente_2)
    self.__corrigir_descendentes()
```

Figura 5: Método de reprodução da população

Após reproduzir a população, executa-se o método `mutar_populacao` para inserir mutações na população de acordo com a probabilidade de ocorrência de mutação definida pelo usuário. Neste método, a abordagem utilizada, também seguiu a adotada por Alves e Pinto, onde uma casa aleatória é selecionada do indivíduo para ser trocada com qualquer outra casa do mesmo indivíduo que possua um vizinho válido da casa selecionada. No artigo utilizado como referência, este operador foi denominado como **Operador de Mutação Baseada no Vizinho Válido**.

```
def mutar_populacao(self):
    """
    Operador de mutacao baseado na troca de uma casa com
    outra que possua um vizinho pertencente a seu conjunto
    troca por vizinho valido
    """

    movimentos_legais = deepcopy(self.movimentos_legais)

    for individuo in self.populacao + self.descendentes:
        p = random.random()
        if p < self.probabilidade_mutacao:
            c = random.choice(individuo) # uma casa qualquer do individuo
            for i in range(0, self.tamanho_individuo - 1):
                # quando alguem possui mesma vizinho de c e nao e c
                if individuo[i + 1] in movimentos_legais[c] \
                    and individuo[i] is not c:
                    individuo = list(individuo)
                    individuo[i], c = c, individuo[i]
                    individuo = tuple(individuo)
```

Figura 6: Método para aplicar mutações na população

Uma vez a população estando mutada ela é novamente avaliada, porém agora, junto a seus descendentes e então, ela é renovada, no método `renovar_populacao` onde a nova população é setada para conter os 50 indivíduos mais aptos dos pais somados aos 50 indivíduos mais aptos dos descendentes. Então, uma geração é finalizada e os testes para finalização da execução do AG são realizados – se a geração chegou ao limite máximo de gerações ou se o melhor elemento da população é solução ótima ($\text{fitness} = n^2 - 1$) para o problema.

```
def renovar_populacao(self):
    """
    metodo para renovacao de individuos da populacao segundo criterio
    metade da populacao corrente mais apta e metade dos descendentes
    mais aptos
    """

    sort_criteria = lambda x, y: -1 if self.fitness(x) < self.fitness(y) \
        else 1 if self.fitness(x) > self.fitness(y) \
        else 0

    self.populacao.sort(sort_criteria, reverse=True)
    self.descendentes.sort(sort_criteria, reverse=True)

    self.populacao = self.populacao[0:self.tamanho_populacao/2] + \
        self.descendentes[0:self.tamanho_populacao/2]
    self.descendentes = []
    self.populacao.sort(sort_criteria, reverse=True)
```

Figura 7: Método para renovar população

```
# teste de condicao para estado Otimo
if self.fitness(self.populacao[0]) == (self.tamanho_individuo - 1):
    Otimo = True
    break
```

Figura 8: Teste de condição para alcance de estado ótimo

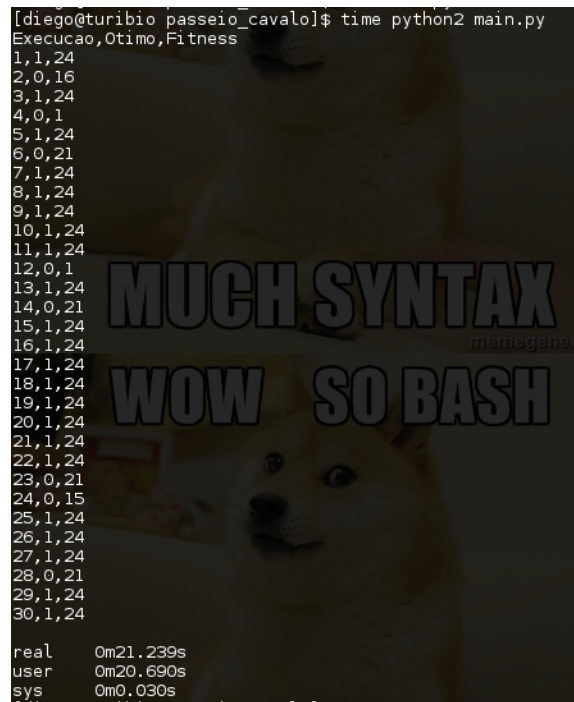
Execuções e resultados obtidos:

Os seguintes parâmetros foram utilizados na execução do algoritmo genético:

- raiz quadrada do tamanho do tabuleiro (n)
- tamanho da população
- numero maximo de gerações
- probabilidade de mutação

Não foi admitida a possibilidade de não ocorrer reproduções, por isso, a probabilidade de reprodução considerada foi de 100% e por isso, não existe um parâmetro probabilidade de reprodução no construtor do algoritmo genético. Os parâmetros tamanho_populacao, geracoes e probabilidade_mutacao tiveram seus valores fixados para qualquer caso em 40, 20 e 0.25 respectivamente. O único parâmetro que foi variado é a raiz quadrada do tamanho do tabuleiro, que variou entre os seguintes valores: 5, 8, 16 e 32. Não foram testados tabuleiros de maiores dimensões que essas, pois, para um tabuleiro de dimensão 64, a execução demorou 2 minutos para ocorrer e em dimensões superiores, já passava dos 15 minutos, não sendo por isso factíveis os testes para tais dimensões.

Para 30 execuções quando $n = 5$ os seguintes resultados foram obtidos:



```
[diego@turibio passeio_cavalo]$ time python2 main.py
Execucao,Otimo,Fitness
1,1,24
2,0,16
3,1,24
4,0,1
5,1,24
6,0,21
7,1,24
8,1,24
9,1,24
10,1,24
11,1,24
12,0,1
13,1,24
14,0,21
15,1,24
16,1,24
17,1,24
18,1,24
19,1,24
20,1,24
21,1,24
22,1,24
23,0,21
24,0,15
25,1,24
26,1,24
27,1,24
28,0,21
29,1,24
30,1,24

real    0m21.239s
user    0m20.690s
sys      0m0.030s
```

Figura 9: 30 execuções para tabuleiro de tamanho 5

A primeira coluna significa o número da execução, a segunda se a execução do ag conseguiu encontrar uma solução ótima → 1 se sim, 0 se não e a terceira mostra o fitness da solução encontrada, 24 para quando a solução é ótima: $24 = 5^2 - 1$ e qualquer número diferente de 24 quando a solução ótima não é encontrada.

Nessa execução, o valor ótimo foi encontrado 22 vezes e as piores execuções de todas foram a 4ª e a

12ª onde o valor de fitness encontrado foi igual a 1. O valor médio encontrado foi de 21.5 e o desvio padrão foi de 5.9203. Esses dados encontram-se melhores representados na tabela a seguir

Tabela 1 – Cálculos de Máximo, Mínimo Média e Desvio Padrão para 30 execuções sintetizadas quando $n = 5$

Mínimo	Máximo	Média	Desvio Padrão
1	24	21.5000	5.9203

Como mencionado previamente, somente tabuleiros de tamanho até 32 foram factíveis em termos de tempo computacional. Então, foram calculados valores de fitness para tabuleiros de tamanho 5, 8, 16 e 32 e os resultados obtidos encontram-se na tabela abaixo:

Tabela 2 – Cálculos de Máximo, Mínimo Média e Desvio Padrão e tempo de execução para 30 execuções sintetizadas quando $n = 5, 8, 16$ e 32

n	min	max	media	desvio_padrao	tempo (s)
5	0	24	22.1	5.9741	16.99
8	63	63	63	0	27.89
16	255	255	255	0	117.13
32	1023	1023	1023	0	1444.43

Análise dos resultados:

Pode-se considerar muito estranho o fato de que para indivíduos de maior tamanho a solução obtida pelo algoritmo genético seja ótima em todas as execuções, mas isso tem uma justificativa. A seleção que ocorre pelo método da roleta, atribui a cada um dos indivíduos da população, uma posicao na roleta baseada no somatorio dos fitness dos indivíduos anteriores a ele, até o momento dele ser processado e existe uma disparidade muito grande entre o fitness dos indivíduos gerados pela regra de warnsdorff com os indivíduos aleatórios para tabuleiros cujo n seja maior que 5. Um indivíduo quando é gerado pela regra de warnsdorff, tende a ter um fitness bem próximo do valor ótimo de resolução do problema:

$n^2 - 1$ e o fitness de indivíduos gerados pela abordagem aleatória fica próximo de valores entre 0 e 4. Por isso, os indivíduos de warnsdorff ocupam fatias bem maiores na roleta do que indivíduos aleatórios, vista a disparidade elicitada. Uma possível “solução” (não é desejável, pois foi muito interessante o AG chegar no valor ótimo em todas as vezes) seria diminuir a inserção de indivíduos warnsdorff na população inicial. Hoje em uma população de 100 indivíduos por exemplo, 20 são warnsdorff.

Conclusões:

Pode-se verificar o sucesso da implementação do algoritmo genético para a resolução dos problemas visto que a valor ótimo foi encontrado em todas as execuções. E um sucesso maior ainda ocorre, uma vez que a média das soluções encontradas quando $n = 5$ foi 22.1, bem próximo ao valor ótimo de 24 e que nas outras execuções, quando $n \geq 8$, não existiu desvio padrão e o valor ótimo foi encontrado em cada uma das 30 execuções. Isso só permite concluir que o algoritmo genético implementado funcionou com excelência.

Referências:

- TAMBERLINI ALVES, F.; PINTO, P. E. D - Aplicação de Algoritmos Genéticos ao Problema do Percurso do cavalo. Disponível em: <<http://www.e-publicacoes.uerj.br/index.php/cadinf/article/viewFile/6556/4674>>. Acesso em: 06 Jun. 2016
- PYTHON. Python 2.7.11 documentation. Disponível em: <<https://docs.python.org/2/>>. Acesso em: 10 Jun. 2016