

Primeira lista de exercícios

Alberto Romanhol Moreira - 2017051564

Problema N-Queens

Para realização do trabalho, utilizou-se de um programa desenvolvido em python. Ao rodar o programa, é solicitada o número de rainhas, *n-queens*, para se procurar a solução via algoritmo genérico. A partir desse número de rainhas, é definido o tamanho da população e o número máximo de gerações.

São criados cromossomos, em forma de vetor de tamanho N, contendo número de 1 a N. Sendo estes, representando a linha em que uma rainha se encontra e a coluna representado pela posição deste no vetor.

A partir da definição de uma função de *fitness*, são realizadas operações genéticas de seleção de parentes e de mutações em cima dos filhos. Isso ocorre até se encontrar o melhor fitness ou haver limite de gerações.

Bibliotecas importadas

Primeiro, importa-se as bibliotecas de terceiros que serão utilizadas no desenvolvimento do trabalho. Sendo:

- A **random** para ser geradora de número aleatórios;
- A **numpy** para fazer manipulação em array.

In []:

```
import random as rnd
import numpy as np
```

Geração inicial de indivíduos

É feita uma função responsável para geração inicial dos indivíduos, as soluções candidatas da primeira geração, representada pela função:

In []:

```
def genotype_queens(n):
    return [ rnd.randint(1, n) for _ in range(n)]
```

Função *fitness*

Desenvolve-se uma função de *fitness*, para avaliar a performance de uma função candidata. Possui alguns pontos a se destacar.

- A partir do cromossomos de rainhas, reduz-se em uma unidade as linhas em que as rainhas se encontram, para o valor ser condizente ao mostrado pela função for (que se inicia em zero);
 - Para cada coluna, compara-se o valor presente nas outras colunas, a se encontrar uma forma de ataque, para outras rainhas:
 - Se está na mesma coluna, a partir dos valores da coluna.
 - Se está na mesma linha, a partir dos valores da linha.
 - Se ocorre ataque diagonal. A partir da inclicação entre a coluna e linha relativa.
 - Caso o ataque não acontece, é somado uma unidade ao score, contabilizando que não ocorreu o ataque;
 - Ao final, o score é dividido por dois, a fim de se evitar duplicidades.

```
In [ ]: def fitness(chromosome):
        score = 0
        n = len(chromosome)
        chromosome = np.array(chromosome) - 1

        for col in range(n):
            row = chromosome[col]

            for other_col in range(n):
                other_row = chromosome[other_col]
                if other_col == col:
                    continue
                if other_row == row:
                    continue
                if other_col + other_row == col + row:
                    continue
                if other_col - other_row == col - row:
                    continue
                score += 1
        return score/2
```

Mecanismo de seleção

Defini-se então, dois mecanismos de seleção.

- Um sendo uma forma de torneio. Retornando o maior candidato em uma tamanho de escolha aleatória da população.

```
In [ ]: def selected_tournament(population):
        n = len(population)
        return bigger_fitness_genotype(population[rnd.randint(0, n-1):n])
```

- O outro sendo uma roleta russa, definida a partir da soma total de chance de fitness da população. Escolhe-se então, um candidato aleatório desta população.

```
In [ ]: def roulette_wheel(population):
```

```

probabilities = [probability(n) for n in population]

total = sum(probabilities)
pick = rnd.uniform(0, total)
current = 0

for i in range(len(probabilities)):
    current += probabilities[i]
    if current > pick:
        return i

```

- Função que retorna a probabilidade do indivíduo ser o melhor.

```

In [ ]: def probability(chromosome):
        return fitness(chromosome) / max_collisions

```

Cria-se então, a função que será responsável por escolher o método a ser utilizado, com uma chance de 20% para o torneio.

```

In [ ]: def pick_parent(population):
        selected_tournament_probability = 0.2

        if (selected_tournament_probability > rnd.random()):
            parent = selected_tournament(population)
        else:
            parent = population[roulette_wheel(population)]

        return parent

```

Variação genética

Define-se então, dois mecanismos de variação genética.

- Uma função de crossover, que divide os parentes em posição aleatória de 0 a N. E une as diferentes partes.

```

In [ ]: def crossover(x, y):
        n = len(x)
        c = rnd.randint(0, n - 1)
        return x[0:c] + y[c:n], y[0:c] + x[c:n]

```

- Uma função de mutação, que a partir da chance de mutação de 5%, pega uma posição aleatória no cromossomo e define um valor aleatório de 0 a N para esta posição.

```

In [ ]: def mutate(chromosome):
        mutation_probability = 0.05

        if (mutation_probability > rnd.random()):
            n = len(chromosome)
            position = rnd.randrange(0, n)
            chromosome[position] = rnd.randrange(1, n + 1)

        return chromosome

```

É definido então a função responsável pela geração do filho, a partir dos pais. A partir destes, escolhe-se o de maior fitness para compor a nova população.

```
In [ ]: def pick_children(parent1, parent2):
        child1, child2 = crossover(parent1, parent2)
        child1 = mutate(child1)
        child2 = mutate(child2)

        child = bigger_fitness_genotype([child1, child2])

        return child
```

Criação de uma nova população

A partir das definições de seleção e variação genética, define-se uma função responsável pela geração de uma nova população. A partir dos métodos já citados de `pick_parent` e `pick_children`.

```
In [ ]: def new_generation(population):
        new_population = []

        for _ in range(len(population)):
            parent1 = pick_parent(population)
            parent2 = pick_parent(population)
            child = pick_children(parent1, parent2)
            #show_children(child)
            new_population.append(child)

            if fitness(child) == max_collisions: break

        return new_population
```

Além disso, utiliza-se de uma função para exibir o cromossomo no console e o seu respectivo fitness. Função foi comentada para melhor entrega do trabalho.

```
In [ ]: def show_children(child):
        print('chromosome = {}, fitness = {}'.format(str(child), fitness(child)))
```

Encontrando a melhor população

É definido uma função que irá encontrar a melhor população, que possui como critério de parada o número máximo de gerações ou quando encontra-se o *fitness* na presente geração.

```
In [ ]: def find_best_population(population):
        generation = 0

        max_generation = get_max_generation()

        while generation < max_generation and not max_collisions in [fitness(queens) for
```

```

        generation += 1
        population = new_generation(population)
        print('generation = {}, best fitness = {}'.format(generation, (max([fitness(
        print('found a generation', generation)
    return population

```

A função de número máximo de gerações é definida a partir do número de gerações. A partir da função linear $-13 \cdot \text{len}(\text{pythonpopulation}[0]) + 277$. Esta definida, para o limite de geração para 6 rainhas ser de aproximadamente 200 e o de 20, ser 20, valores definidos a partir de testes com o código.

```

In [ ]: def get_max_generation():
        max_generation = -13*len(population[0]) + 277
        print('max_generation = {}'.format(max_generation))
        return max_generation

```

Funções auxiliares

Define-se ainda, algumas funções auxiliares, como a para retornar o indivíduo com o melhor fitness da população.

```

In [ ]: def bigger_fitness_genotype(population):
        best_fitness = 0

        for chrom in population:
            if fitness(chrom) > best_fitness:
                best_fitness = fitness(chrom)
                c = chrom

        return c

```

```

In [ ]: def bigger_fitness_genotype(population):
        best_fitness = 0

        for chrom in population:
            if fitness(chrom) > best_fitness:
                best_fitness = fitness(chrom)
                c = chrom

        return c

```

E uma outra função para transformar o vetor das rainhas em um board, sendo x representando um espaço em branco e o Q representando o posicionamento da rainha.

```

In [ ]: def print_board(n_queens):
        board = []
        n = len(n_queens)
        for _ in range(n):
            board.append(["x"] * n)
        for i in range(n):
            board[n_queens[i]-1][i]="Q"
        for row in board:
            print (" ".join(row))

```

Programa principal

A partir da entrada do número de rainhas a ser utilizado no problema, calcula-se o número máximo de colisões que pode acontecer $(n_queens * (n_queens - 1)) / 2$. Defini-se ainda, a partir do número de rainhas, o tamanho da população de $2 * (n_queens ** 2)$.

A partir da geração de indivíduos, gera-se valores do tamanho da população e de acordo com o número de rainhas. A partir das funções definidas anteriormente, é encontrado a melhor população, seguindo o critério de *fitness* ou de parada por número de gerações. E, a partir disso, é encontrado o indivíduo com maior *fitness*, não sendo necessariamente o máximo.

```
In [ ]: n_queens = int(input("number of queens: "))
max_collisions = (n_queens*(n_queens-1))/2
print('max fitness:', max_collisions)

n_population = 2*(n_queens**2)
print('n_population:', n_population)

population = [genotype_queens(n_queens) for _ in range(n_population)]

best_population = find_best_population(population)
bigger_fitness = bigger_fitness_genotype(best_population)

max fitness: 28.0
n_population: 128
max_generation = 173
generation = 1, best fitness = 25.0
generation = 2, best fitness = 25.0
generation = 3, best fitness = 26.0
generation = 4, best fitness = 27.0
generation = 5, best fitness = 27.0
generation = 6, best fitness = 27.0
generation = 7, best fitness = 27.0
generation = 8, best fitness = 27.0
generation = 9, best fitness = 27.0
generation = 10, best fitness = 27.0
generation = 11, best fitness = 27.0
generation = 12, best fitness = 27.0
generation = 13, best fitness = 28.0
found a generation 13
```

A partir desse resultado, pode-se imprimir no console o resultado do programa. Apresentando os dados do problema, o tipo de solução encontrada, o fitness do indivíduo e uma representação do tabuleiro.

```
In [ ]: print('-----')
print('n queen problem - genetic algorithm')
print('n_queens:', n_queens)
print('n_population:', n_population)
print('max fitness:', max_collisions)
print('-----')
if fitness(bigger_fitness) == max_collisions:
    print('found a solution by fitness')
```

```
else:
    print('no solution, stopped by generation limit')
    print('found fitness:', fitness(bigger_fitness))
    print('best_queen:', bigger_fitness)
    print_board(bigger_fitness)
    print('-----')
```

```
-----
n queen problem - genetic algorithm
n_queens: 8
n_population: 128
max fitness: 28.0
-----
found a solution by fitness
found fitness: 28.0
best_queen: [6, 3, 7, 4, 1, 8, 2, 5]
x x x x Q x x x
x x x x x x Q x
x Q x x x x x x
x x x Q x x x x
x x x x x x x Q
Q x x x x x x x
x x Q x x x x x
x x x x x Q x x
-----
```