n_queens

March 30, 2024

1 Introdução

Neste trabalho procuramos criar um algoritmo **Algoritmo Evolucionário** para a resolução do problema das **N Rainhas**. O processo de descrição da construção do algoritmo, bem como as escolhas de parâmetros, serão descritos em detalhes para cada sessão do algoritmo.

A forma do algoritmo final se baseia no pseudocódigo fornecido na descrição do trabalho, sendo ele o seguinte:

BEGIN

```
INITIALISE population with random candidate solutions;

EVALUATE each candidate;

REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
OD
```

END

As três principais referências utilizadas na elaboração deste trabalho foram: - Introduction to evolutionary computing, A.E. Eiben and J.E. Smith, Springer, 2015. - Permutation-Based Evolutionary Algorithms for Multidimensional Knapsack Problems, Jens Gottlieb, Proceedings of the 2000 ACM symposium on Applied computing-Volume 1 - Manual de Computação Evolutiva e Metaheurísticas, Antônio Gaspar-Cunha; Ricardo Takahashi; Carlos Henggeler Antunes, Editora UFMG, 2013.

[ ]:
```
import random as rd
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
from pandas import Series
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
```

2 Criação da População e Parâmetros

Para iniciar a **população**, por meio da representação exigida no relatório, será gerada aleatoriamente uma população de tamanho decidido arbitrariamente, em que cada indivíduo possui um número n de rainhas (uma para cade linha do tabuleiro).

[ ]:
```
crossover_rate = 1
mutation_rate = 0.8
max_generations = 10000

def init_population(_mu:int = 20, n:int = 8):
    population = []
    for i in range (_mu):
        population.append(rd.sample(range(n), n))
    return population
```

3 Função de Aptidão

Definimos a **aptidão** como o número de xeques que as rainhas estão expostas. Vale notar que o objetivo do problema é **minimizar** a função de aptidão, para termos o menor número de xeques possíveis e, se possível, nenhum.

[ ]:
```
def fitness_nq(solution):
    xeques = 0
    for i in range(0,len(solution)):
        for j in range(0,len(solution)):
            if i!=j:
                if i-solution[i] == j-solution[j] or i+solution[i] ==␣
 ↪j+solution[j]:
                    xeques+=1
    return xeques
```

4 Mutação da Solução (Swap)

Realizamos a **mutação** utilizando a técnica de **Swap**. Para isso, escolhemos aleatoriamente duas posições do vetor de representação do indivíduo e trocamos seus valores. > **Exemplo:** Caso tenhamos escolhido as posições 2 e 4 para o vetor [0,1,2,3,4,5,6,7], teremos a mutação [0,1,4,3,2,5,6,7].

[ ]:
```
def mutate_solution_swap(solution):
    to_swap_lines = rd.sample(range(0,len(solution)), 2)
```

```
    solution[to_swap_lines[0]], solution[to_swap_lines[1]] =␣
↪solution[to_swap_lines[1]], solution[to_swap_lines[0]]
```

5 Mutação da Solução (Insert)

Realizamos a **mutação** utilizando a técnica de **Insert**. Para isso, escolhemos aleatoriamente um gene da solução candidata e o movemos para uma nova posição aleatória. > **Exemplo:** Caso tenhamos escolhido as posições o gene 6 para a posição do vetor [0,1,2,3,4,5,6,7], teremos a mutação [0,1,6,2,3,4,5,7].

[ ]:
```
def mutate_solution_insert(solution):
    line = rd.choice(range(0,len(solution)))
    gene = rd.choice(solution)
    gene_current_index = solution.index(gene)
    solution.pop(gene_current_index)
    solution.insert(line, gene)
```

6 Inversão

Este é um operador pouco utilizado em contextos práticos, porém foi proposto por John Holland em seu livro "Adaptation in Natural and Artificial Systems".

Realizamos a **inversão** escolhendo aleatoriamente duas posições do vetor de representação do indivíduo e invertendo os valores entre elas.

> **Observação:** A inversão pode ser considerada como um *crossover* de uma solução consigo mesma.

[ ]:
```
def inversion(pop):
    inversion_rate = 0.1

    for solution in pop:
        if rd.random() < inversion_rate:
            to_swap_lines = rd.sample(range(0,len(solution)), 2)
            x1, x2 = min(to_swap_lines), max(to_swap_lines)
            solution[x1:x2] = solution[x1:x2][::-1]
```

7 Seleção dos Pais

Para a **seleção dos pais**, utilizamos um método de **Torneio** alternativo. Neste método, escolhemos aleatoriamente cinco indivíduos da população e selecionamos os dois que possuem a menor função de aptidão como pais.

8 Recombinação (Cut-and-crossfill)

Realizamos a **recombinação** utilizando a técnica de **Cut-and-crossfill**. Esse método irá criar dois filhos, cada um deles irá possuir uma parte inicial do vetor de um pai e irá 'retirar' os outros valores, que ainda não possui, do outro pai.

> Obs: As explicações de cada parte da técnica utilizada podem ser encontradas nos comentários dentro da própria função presente abaixo

[ ]:
```
def recombine_cut_and_crossfil(population):
    # Escolha de 5 possíveis pais aleatoriamente
    random_possible_parents = [population[i] for i in rd.
 ↪sample(range(0,len(population)), 5)]

    # Melhores duas possíveis soluções para pais
    best_possible_parents_indexes = np.argpartition([fitness_nq(sol) for sol in␣
 ↪random_possible_parents] ,2)[:2]
    parent_1 = random_possible_parents[best_possible_parents_indexes[0]]
    parent_2 = random_possible_parents[best_possible_parents_indexes[1]]

    # Ter ao menos 1 elemento de cada pai para cada filho e preencher os filhos␣
 ↪(inicialmente)
    first_section_cut_index = rd.choice(range(0, len(parent_1)-1))

    child_1 = np.zeros(len(parent_1))
    child_2 = np.zeros(len(parent_2))

    child_1[:first_section_cut_index+1] = parent_1[:first_section_cut_index+1]
    child_2[:first_section_cut_index+1] = parent_2[:first_section_cut_index+1]

    # Preenchendo o resto dos vetores filhos com as soluções dos próximos pais
    current_child_index = first_section_cut_index

    for value in parent_2:
        if current_child_index >= len(child_1)-1:
            break
        if value in child_1:
            continue

        child_1[current_child_index+1] = value

        current_child_index += 1


    current_child_index = first_section_cut_index

    for value in parent_1:
        if current_child_index >= len(child_2)-1:
```

```
            break
        if value in child_2:
            continue

        child_2[current_child_index+1] = value

        current_child_index += 1

    child_1 = [int(x) for x in child_1]
    child_2 = [int(x) for x in child_2]

    return child_1, child_2
```

9 Recombinação (Uniform order based crossover)

Na técnica de **Uniform order based crossover** sugerida por Davis em "Handbook of Genetic Algorithms", tenta simular a técnica de recombinação probabilistica em representações binárias. Para isso, é realizada a escolha de um gene para acrescentar ao filho a partir do primeiro pai com uma probabilidade de 50%. Ao terminar de selecionar os genes do primeiro pai, os genes restantes são adicionados ao filho a partir do segundo pai.

[ ]:
```
def recombine_uniform_order_based_crossover(population):
    # Escolha de 5 possíveis pais aleatoriamente
    random_possible_parents = [population[i] for i in rd.
 ↪sample(range(0,len(population)), 5)]

    # Melhores duas possíveis soluções para pais
    best_possible_parents_indexes = np.argpartition([fitness_nq(sol) for sol in␣
 ↪random_possible_parents] ,2)[:2]
    parent_1 = random_possible_parents[best_possible_parents_indexes[0]]
    parent_2 = random_possible_parents[best_possible_parents_indexes[1]]

    # Criação dos filhos vazios

    child_1 = np.zeros(len(parent_1))
    child_2 = np.zeros(len(parent_2))

    child_1_index = 0
    child_2_index = 0

    # Escolhendo os elementos a serem inseridos nos filhos
    for value in parent_1:
        if rd.random() < 0.5:
            child_1[child_1_index] = value

            child_1_index += 1
```

```
    for value in parent_2:
        if rd.random() < 0.5:
            child_2[child_2_index] = value

            child_2_index += 1

    # Preenchendo os filhos com os elementos restantes
    for value in parent_2:
        if value not in child_1:
            child_1[child_1_index] = value
            child_1_index += 1

    for value in parent_1:
        if value not in child_2:
            child_2[child_2_index] = value
            child_2_index += 1

    child_1 = [int(x) for x in child_1]
    child_2 = [int(x) for x in child_2]

    return child_1, child_2
```

10 Escolha de Mutações

Realizamos com **80%** de chance a **mutação** para cada um dos dois filhos.

[ ]:
```
def mutate_childs(child_1, child_2, mutation_operator):
    # Mutações em 80% das vezes para cada filho

    if rd.uniform(0, 1) < mutation_rate:
        mutation_operator(child_1)

    if rd.uniform(0, 1) < mutation_rate:
        mutation_operator(child_2)
```

11 Seleção de Candidatos

A **população** que continuará será referente aos n-2 indivíduos, após a inserção dos 2 filhos, que melhor se adaptam ao problema, ou seja, os 2 indivíduos com os piores valores de aptidão serão retirados da população.

[ ]:
```
def select_new_population(pop, child_1, child_2):
    # Retirando piores duas soluções da nova população

    pop.append(child_1)
```

```
    pop.append(child_2)

    elements_indexes_to_pop = np.argpartition([fitness_nq(sol) for sol in pop],␣
↪-2)[-2:]

    for index in sorted(elements_indexes_to_pop, reverse=True):
        del pop[index]

    return pop
```

12 Encontrar a Solução

Uma função que nos permite realizar o processo de um **Algoritmo Evolucionário** de forma simples, sendo necessário escolher apenas a quantidade de indivíduos na população e quantas rainhas devem existir no tabuleiro. O formato do código se baseia no pseudocódigo descrito na Introdução.

[ ]:
```
def find_best_solution(_mu, n, recombine_operator, mutation_operator,␣
↪inversion_operator=None):
    pop = init_population(_mu, n)

    current_generation = 0
    fitness_pop_vectors = []

    while True:
        fitness_pop = [fitness_nq(sol) for sol in pop]
        fitness_pop_vectors.append(min(fitness_pop))

        if 0 in fitness_pop or current_generation >= max_generations:
            break

        if inversion_operator is not None:
            inversion_operator(pop)

        child_1, child_2 = recombine_operator(pop)

        mutate_childs(child_1, child_2, mutation_operator)

        pop = select_new_population(pop, child_1, child_2)

        current_generation += 1

    best_solution = pop[np.argmin([fitness_nq(sol) for sol in pop])]

    best_solution_fitness = fitness_nq(best_solution)
```

```
    return best_solution, best_solution_fitness, current_generation,␣
 ↪fitness_pop_vectors
```

[ ]:
```
mutation_operators = [mutate_solution_swap, mutate_solution_insert]
inversion_operators = [inversion, None]
crossover_operators = [recombine_cut_and_crossfil,␣
 ↪recombine_uniform_order_based_crossover]

initial_population_values = [10, 20, 50]
mutation_operators_names = ["Swap", "Insert"]
inversion_operators_names = ["Inversion", "Não Aplicado"]
crossover_operators_names = ["Cut and Crossfil", "Uniform Order Based␣
 ↪Crossover"]

operators_possibilites_tuple = list(product(enumerate(mutation_operators),␣
 ↪enumerate(inversion_operators), enumerate(crossover_operators)))

operators_possibilites = [[x[1] for x in possibility] for possibility in␣
 ↪operators_possibilites_tuple]
operators_possibilites_names = [[mutation_operators_names[possibility[0][0]],␣
 ↪inversion_operators_names[possibility[1][0]],␣
 ↪crossover_operators_names[possibility[2][0]]] for possibility in␣
 ↪operators_possibilites_tuple]
```

[ ]:
```
analysis = dict()

operator_possibility = 0
for mutation_operator, inversion_operator, crossover_operator in␣
 ↪operators_possibilites:
    generations_to_find_best_solution = []
    for i in range(1000):
        _, _, current_generation, _ = find_best_solution(20, 8,␣
 ↪crossover_operator, mutation_operator, inversion_operator)
        generations_to_find_best_solution.append(current_generation)
    generations_to_find_best_solution =␣
 ↪Series(generations_to_find_best_solution)

    analysis[
        f'Mutação: {operators_possibilites_names[operator_possibility][0]} -␣
 ↪Inversão: {operators_possibilites_names[operator_possibility][1]} -␣
 ↪Recombinação: {operators_possibilites_names[operator_possibility][2]}'] =␣
 ↪generations_to_find_best_solution.describe()

    operator_possibility += 1

for key, value in analysis.items():
```

```
    print(f'{key}:')
    print("Média: ", value['mean'])
    print("Desvio Padrão: ", value['std'])
    print('\n\n')
```

```
Mutação: Swap - Inversão: Inversion - Recombinação: Cut and Crossfil:
Média:  59.018
Desvio Padrão:  53.809210244224005


Mutação: Swap - Inversão: Inversion - Recombinação: Uniform Order Based
Crossover:
Média:  75.199
Desvio Padrão:  76.05737389052503


Mutação: Swap - Inversão: Não Aplicado - Recombinação: Cut and Crossfil:
Média:  145.408
Desvio Padrão:  240.82595515574457


Mutação: Swap - Inversão: Não Aplicado - Recombinação: Uniform Order Based
Crossover:
Média:  166.965
Desvio Padrão:  160.36002966485805


Mutação: Insert - Inversão: Inversion - Recombinação: Cut and Crossfil:
Média:  63.009
Desvio Padrão:  62.72611485246378


Mutação: Insert - Inversão: Inversion - Recombinação: Uniform Order Based
Crossover:
Média:  70.958
Desvio Padrão:  68.26044040089194


Mutação: Insert - Inversão: Não Aplicado - Recombinação: Cut and Crossfil:
Média:  500.219
Desvio Padrão:  1344.793075329605
```

```
Mutação: Insert - Inversão: Não Aplicado - Recombinação: Uniform Order Based
Crossover:
Média:  169.939
Desvio Padrão:  185.00013061435212
```

13 Conclusão

Algumas conclusões são facilmente retiradas das aplicações aqui realizadas. A primeira delas é que o **Algoritmo Evolucionário**, em especial o **Algoritmo Genético** é uma técnica muito poderosa para a resolução de problemas complexos, como o das **N Rainhas**.

A utilização do operador de inversão adiciona uma nova possibilidade de exploração do espaço de busca, porém, como esperado, não se mostrou tão eficiente quanto os operadores de mutação e recombinação. Este é o motivo pelo qual ele não é tão utilizado em contextos práticos.

O operador de recombinação **Cut-and-crossfill** se mostrou mais eficiente que o **Uniform order based crossover** para o problema das **N Rainhas**. Tal fato provavelmente decorre de o **Uniform order based crossover** "perder" a informação de posição das rainhas, o que é crucial para a resolução do problema.

Por fim, o operador de mutação que gerou resultados mais satisfatórios foi o **Insert**, seguido pelo **Swap**. Isso provavelmente se deve ao fato de que o **Insert** permite uma maior exploração do espaço de busca, o que é crucial para a resolução do problema das **N Rainhas**.