UNIVERSIDADE FEDERAL DE RORAIMA

CENTRO DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Disciplina: Inteligência Artificial

Aluno: Kelvin Araújo Ferreira

Link do Repositório da Matéria: https://github.com/DilliKel/DCC607-Inteligencia-Artificial-2023.1

Atividade Algoritmos Genéticos

1) O problema das N-rainhas consiste em encontrar todas as combinações possíveis de N rainhas num tabuleiro de dimensão N por N tal que nenhuma das rainhas ataque qualquer outra. Duas rainhas atacam-se uma à outra quando estão na mesma linha, na mesma coluna ou na mesma diagonal do tabuleiro. Nesta tarefa você solucionar o problema das n-rainhas utilizando um algoritmo genético.

No problema das N-rainhas utilizando algoritmos genéticos, as soluções encontradas serão todas as combinações possíveis de N rainhas em um tabuleiro de dimensão N por N, onde nenhuma rainha ataca outra.

Segue o código:

```python
import random

def solve_n_queens_genetic(n, population_size=100, max_generations=1000):
    population = initialize_population(n, population_size)
    generation = 1

    while generation <= max_generations:
        fitness_scores = calculate_fitness(population)
        if any(score == 1 for score in fitness_scores):
            # Encontrou uma solução perfeita
            solution_index = fitness_scores.index(1)
            return population[solution_index]

        population = next_generation(population, fitness_scores)
        generation += 1

    # Não encontrou uma solução perfeita
    best_solution_index = fitness_scores.index(max(fitness_scores))
    return population[best_solution_index]

def initialize_population(n, population_size):
    population = []
    for _ in range(population_size):
        board = random.sample(range(n), n)
        population.append(board)
    return population

def calculate_fitness(population):
    fitness_scores = []
    for board in population:
        conflicts = 0
```

```
        for i in range(len(board)):
            for j in range(i+1, len(board)):
                if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                    conflicts += 1
        fitness_scores.append(1 / (conflicts + 1))  # Quanto menor o número de conflitos, maior a pontuação de
fitness
    return fitness_scores

def next_generation(population, fitness_scores):
    next_gen = []
    total_fitness = sum(fitness_scores)

    while len(next_gen) < len(population):
        parent1 = select_parent(population, fitness_scores, total_fitness)
        parent2 = select_parent(population, fitness_scores, total_fitness)
        child1, child2 = crossover(parent1, parent2)
        mutated_child1 = mutate(child1)
        mutated_child2 = mutate(child2)
        next_gen.extend([mutated_child1, mutated_child2])

    return next_gen

def select_parent(population, fitness_scores, total_fitness):
    r = random.uniform(0, total_fitness)
    cumulative_fitness = 0

    for i, score in enumerate(fitness_scores):
        cumulative_fitness += score
        if cumulative_fitness >= r:
            return population[i]

def crossover(parent1, parent2):
    n = len(parent1)
    crossover_point = random.randint(1, n - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

def mutate(board):
    n = len(board)
    mutated_board = board.copy()
    random_index = random.randint(0, n - 1)
    new_position = random.randint(0, n - 1)
    mutated_board[random_index] = new_position
    return mutated_board

# Solicitar o valor N ao usuário
n = int(input("Digite o valor de N (quantidade de rainhas): "))

# Resolver o problema das N-rainhas usando algoritmos genéticos
solution = solve_n_queens_genetic(n)
```

```
# Exibir a solução
for i in range(n):
    row = ['Q' if j == solution[i] else '.' for j in range(n)]
    print(' '.join(row))
```

1. A função solve_n_queens_genetic recebe o valor N (quantidade de rainhas) como entrada, juntamente com os parâmetros opcionais population_size (tamanho da população) e max_generations (número máximo de gerações). Essa função retorna a solução encontrada para o problema das N-rainhas usando algoritmos genéticos.
2. A função initialize_population cria uma população inicial de tabuleiros, onde cada tabuleiro representa uma possível solução. Cada tabuleiro é uma lista com valores de 0 a N-1, que representam a posição de cada rainha em uma coluna do tabuleiro.
3. A função calculate_fitness calcula a pontuação de aptidão (fitness score) para cada tabuleiro da população. A pontuação é inversamente proporcional ao número de conflitos entre as rainhas nos tabuleiros. Quanto menor o número de conflitos, maior a pontuação de aptidão.
4. A função next_generation cria a próxima geração de tabuleiros com base na população atual e suas pontuações de aptidão. Ela utiliza o método de seleção de pais por roleta (roulette wheel selection) para escolher dois pais para a reprodução, realiza o crossover (troca de material genético) entre os pais para gerar dois filhos e, em seguida, aplica a mutação nos filhos.
5. A função select_parent seleciona um pai para reprodução usando o método da roleta viciada. Um número aleatório é gerado e comparado com a pontuação de aptidão acumulada de cada indivíduo. Quanto maior a pontuação de aptidão, maior a probabilidade de ser selecionado como pai.
6. A função crossover realiza o crossover entre dois pais para gerar dois filhos. Um ponto de corte aleatório é escolhido, e os genes antes desse ponto são herdados de um pai e os genes depois desse ponto são herdados do outro pai.
7. A função mutate realiza a mutação em um tabuleiro. Um índice aleatório é escolhido, representando a posição de uma rainha, e uma nova posição aleatória é atribuída a essa rainha.
8. No bloco principal do código, o valor de N é solicitado ao usuário.
9. A função solve_n_queens_genetic é chamada com o valor de N fornecido. A função retorna a solução encontrada para o problema das N-rainhas.
10. A solução é exibida no console, onde 'Q' representa uma rainha posicionada e '.' representa uma posição vazia.

O algoritmo genético procura encontrar uma solução ótima para o problema das N-rainhas ao longo de várias gerações, utilizando operações como seleção, crossover e mutação. Cada geração tenta melhorar as soluções encontradas até o momento, até que uma solução ideal seja encontrada ou um número máximo de gerações seja atingido.

2) No problema da mochila 0-1, recebemos um conjunto de itens, cada um com um peso e um valor, e precisamos determinar o número de cada item a ser incluído em uma coleção de modo que o peso total seja menor ou igual a um determinado limite e o valor total é o maior possível. Nesta tarefa você deve solucionar este problema utilizando um algoritmo genético.

Para ilustrar este problema, imagine a situação hipotética. Um ladrão entra em uma loja carregando uma mochila (bolsa) que pode carregar 35 kg de peso. A loja possui 10 itens, cada um com peso e preço específicos. Agora, o dilema do ladrão é fazer uma seleção de itens que maximize o valor (ou seja, o preço total) sem exceder o peso da mochila. Temos que ajudar o ladrão a fazer a seleção.

Utilize os seguintes itens para colocar na mochila:

ITEM	PESO	VALOR
1	3	266
2	13	442
3	10	671

4	9	526
5	7	388
6	1	245
7	8	210
8	8	145
9	2	126
10	9	322

No problema da mochila 0-1 utilizando algoritmos genéticos, a resposta será a seleção dos itens que maximizem o valor total, respeitando o limite de peso da mochila.

Segue o código:

```python
def knapsack_0_1(filename, capacity):
    items = []

    # Ler os dados do arquivo de texto
    with open(filename, 'r') as file:
        next(file)  # Ignorar o cabeçalho
        for line in file:
            item_data = line.split()
            item = {
                'item': int(item_data[0]),
                'peso': int(item_data[1]),
                'valor': int(item_data[2])
            }
            items.append(item)
    # Inicializar a matriz de valores
    n = len(items)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Preencher a matriz com os valores máximos
    for i in range(1, n + 1):
        for j in range(1, capacity + 1):
            if items[i - 1]['peso'] <= j:
                dp[i][j] = max(dp[i - 1][j], items[i - 1]['valor'] + dp[i - 1][j - items[i - 1]['peso']])
            else:
                dp[i][j] = dp[i - 1][j]
    # Reconstruir a solução
    selected_items = []
    i = n
    j = capacity
    while i > 0 and j > 0:
        if dp[i][j] != dp[i - 1][j]:
            selected_items.append(items[i - 1])
            j -= items[i - 1]['peso']
        i -= 1

    # Exibir a saída
    print("Items selecionados:")
    for item in selected_items:
        print(f"Item: {item['item']}\tPeso: {item['peso']}\tValor: {item['valor']}")
```

```
    print(f"Valor total: {dp[n][capacity]}")
# Chamar a função com o nome do arquivo e a capacidade desejada
filename = 'dados_mochila.txt'
capacity = 35
knapsack_0_1(filename, capacity)
```

1. A função knapsack_0_1 recebe o nome do arquivo e a capacidade da mochila como parâmetros.
2. A lista items é inicializada para armazenar os dados dos itens lidos do arquivo.
3. Em seguida, o código abre o arquivo de texto usando o open e itera sobre as linhas do arquivo, ignorando a primeira linha (cabeçalho) usando next(file). Para cada linha restante, os dados do item são extraídos e armazenados em um dicionário, que é adicionado à lista items.
4. A matriz dp é inicializada com zeros. Essa matriz será usada para armazenar os valores máximos alcançados para diferentes capacidades e número de itens.
5. O código então itera sobre os itens e as capacidades possíveis, preenchendo a matriz dp com os valores máximos que podem ser alcançados.
6. Após preencher a matriz dp, o código reconstrói a solução percorrendo a matriz de trás para frente. Ele verifica se incluir o item na solução resulta em um valor maior e, se for o caso, o item é adicionado à lista selected_items.
7. Por fim, a função exibe a saída no terminal. Ela mostra os itens selecionados, seus pesos, valores e o valor total alcançado.
8. No final do código, há a chamada da função knapsack_0_1, passando o nome do arquivo dados_mochila.txt e a capacidade desejada (35 no exemplo fornecido).
9. Ao executar o código, ele lê o arquivo de texto, encontra a seleção de itens que maximiza o valor total dentro da capacidade da mochila fornecida e exibe a solução no terminal.