Генетический алгоритм для задачи расстановки ферзей на шахматной доске работает похоже на процесс эволюции в биологии, где популяция особей приспосабливается к окружающей среде.

1. Инициализация популяции: Алгоритм начинается с создания начальной популяции решений (расстановок ферзей). В нашем случае, каждое решение представляет собой уникальную расстановку ферзей на шахматной доске.
2. Оценка приспособленности (fitness): Каждая расстановка ферзей оценивается на основе количества пар ферзей, которые находятся под угрозой атаки друг друга. Чем меньше атакующих пар, тем лучше приспособленность этой расстановки.
3. Отбор: Решения с лучшей приспособленностью (меньшим количеством атакующих пар) имеют больше шансов быть выбранными для скрещивания и создания новых поколений.
4. Скрещивание (кроссовер): Выбранные родительские расстановки скрещиваются для создания новых потомков (новых расстановок ферзей). Это происходит путем комбинирования частей родительских расстановок.
5. Мутация: Некоторые потомки могут случайно подвергнуться мутации, где их расстановка изменяется с небольшой вероятностью. Это помогает избежать застревания в локальных оптимумах и исследовать больше пространства решений.
6. Повторение: Новое поколение расстановок ферзей заменяет предыдущее, и процесс отбора, скрещивания и мутации повторяется многократно в цикле (поколениях).
7. Критерий остановки: Алгоритм продолжает эволюцию популяции в течение определенного числа поколений или до достижения определенного уровня приспособленности (оптимального решения).
8. Вывод результата: В конце работы алгоритма возвращается лучшая расстановка ферзей, которая является результатом решения задачи.

Таким образом, генетический алгоритм для задачи расстановки ферзей использует механизмы отбора, скрещивания и мутации, чтобы постепенно улучшать расстановку ферзей и приближаться к оптимальному решению без необходимости перебора всех возможных вариантов.

using System;

namespace Queen

{

class Program

{

static void Main()

{

int boardSize = 8; // Размер шахматной доски

double mutationRate = 0.1; // Вероятность мутации

int populationSize = 50; // Размер популяции

int maxGenerations = 1000; // Максимальное количество поколений

// Создаем объект генетического решения для задачи расстановки ферзей

GeneticQueensSolver solver = new GeneticQueensSolver(boardSize, mutationRate, populationSize);

// Инициализируем начальную популяцию

solver.InitializePopulation();

// Решаем задачу расстановки ферзей

QueensArrangement solution = solver.Solve(maxGenerations);

// Выводим найденное решение

PrintBoard(solution.GetPositions());

}

// Метод для вывода расстановки ферзей на доске

static void PrintBoard(int[] positions)

{

int size = positions.Length;

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

if (positions[j] == i)

Console.Write("Q ");

else

Console.Write(". ");

}

Console.WriteLine();

}

}

}

// Создаем класс для представления расстановки ферзей

class QueensArrangement

{

// Поля класса

private int[] queensPositions; // Массив для хранения позиций ферзей

private int boardSize; // Размер шахматной доски

// Конструктор класса

public QueensArrangement(int size)

{

boardSize = size;

queensPositions = new int[boardSize];

}

// Метод для инициализации начальной случайной расстановки ферзей

public void InitializeRandom()

{

Random random = new Random();

for (int i = 0; i < boardSize; i++)

{

queensPositions[i] = random.Next(0, boardSize);

}

}

// Метод для вычисления оценки (fitness) текущей расстановки ферзей

public int CalculateFitness()

{

int attacks = 0;

// Проверяем каждую пару ферзей на конфликт

for (int i = 0; i < boardSize - 1; i++)

{

for (int j = i + 1; j < boardSize; j++)

{

// Проверяем горизонталь, вертикаль и диагонали

if (queensPositions[i] == queensPositions[j] ||

Math.Abs(queensPositions[i] - queensPositions[j]) == Math.Abs(i - j))

{

attacks++;

}

}

}

// Оценка - количество атакующих пар

return attacks;

}

// Метод для скрещивания двух расстановок

public QueensArrangement Crossover(QueensArrangement partner)

{

Random random = new Random();

int crossoverPoint = random.Next(0, boardSize);

QueensArrangement child = new QueensArrangement(boardSize);

for (int i = 0; i < boardSize; i++)

{

if (i < crossoverPoint)

child.queensPositions[i] = queensPositions[i];

else

child.queensPositions[i] = partner.queensPositions[i];

}

return child;

}

// Метод для мутации расстановки

public void Mutate(double mutationRate)

{

Random random = new Random();

for (int i = 0; i < boardSize; i++)

{

if (random.NextDouble() < mutationRate)

{

queensPositions[i] = random.Next(0, boardSize);

}

}

}

// Метод для получения текущей расстановки

public int[] GetPositions()

{

return queensPositions;

}

}

class GeneticQueensSolver

{

private int boardSize; // Размер шахматной доски

private double mutationRate; // Вероятность мутации

private int populationSize; // Размер популяции

private QueensArrangement[] population; // Популяция расстановок ферзей

public GeneticQueensSolver(int size, double mutationRate, int populationSize)

{

this.boardSize = size;

this.mutationRate = mutationRate;

this.populationSize = populationSize;

this.population = new QueensArrangement[populationSize];

}

// Метод для инициализации начальной популяции

public void InitializePopulation()

{

for (int i = 0; i < populationSize; i++)

{

QueensArrangement arrangement = new QueensArrangement(boardSize);

arrangement.InitializeRandom();

population[i] = arrangement;

}

}

// Метод для поиска оптимальной расстановки ферзей

public QueensArrangement Solve(int maxGenerations)

{

int generation = 0;

while (generation < maxGenerations)

{

QueensArrangement[] newPopulation = new QueensArrangement[populationSize];

// Сортируем текущую популяцию по оценке (fitness)

Array.Sort(population, (x, y) => x.CalculateFitness().CompareTo(y.CalculateFitness()));

// Лучшие 2 расстановки переносятся в новую популяцию без изменений

newPopulation[0] = population[0];

newPopulation[1] = population[1];

// Создаем новые расстановки скрещиванием лучших

for (int i = 2; i < populationSize; i++)

{

QueensArrangement parent1 = SelectParent();

QueensArrangement parent2 = SelectParent();

QueensArrangement child = parent1.Crossover(parent2);

child.Mutate(mutationRate);

newPopulation[i] = child;

}

population = newPopulation;

generation++;

}

// Возвращаем лучшую расстановку из последней популяции

return population[0];

}

// Метод для выбора родителя на основе рулеточной селекции

private QueensArrangement SelectParent()

{

Random random = new Random();

int totalFitness = 0;

foreach (QueensArrangement arrangement in population)

{

totalFitness += arrangement.CalculateFitness();

}

int rand = random.Next(0, totalFitness);

int runningSum = 0;

foreach (QueensArrangement arrangement in population)

{

runningSum += arrangement.CalculateFitness();

if (runningSum > rand)

{

return arrangement;

}

}

return population[random.Next(0, populationSize)];

}

}

}

import random def generate\_board(size): board = [0] \* size for i in range(size): board[i] = random.randint(0, size-1) return board def fitness(board): conflicts = 0 for i in range(len(board)): for j in range(i+1, len(board)): if board[i] == board[j] or board[i]+i == board[j]+j or board[i]-i == board[j]-j: conflicts += 1 return conflicts def cross\_over(board1, board2): size = len(board1) index = random.randint(1, size-2) child1 = board1[:index] + board2[index:] child2 = board2[:index] + board1[index:] return child1, child2 def mutate(board): size = len(board) index = random.randint(0, size-1) new\_pos = random.randint(0, size-1) board[index] = new\_pos return board def genetic\_algorithm(board\_size): population\_size = 20 elite\_size = 2 mutation\_rate = 0.1 board\_population = [generate\_board(board\_size) for i in range(population\_size)] for i in range(1000): fitness\_scores = [(fitness(board), board) for board in board\_population] fitness\_scores.sort() best\_boards = [board for score, board in fitness\_scores[:elite\_size]] if fitness\_scores[0][0] == 0: return fitness\_scores[0][1] new\_population = best\_boards while len(new\_population) < population\_size: parent1 = random.choice(best\_boards) parent2 = random.choice(best\_boards) child1, child2 = cross\_over(parent1, parent2) if random.random() < mutation\_rate: child1 = mutate(child1) if random.random() < mutation\_rate: child2 = mutate(child2) new\_population.append(child1) new\_population.append(child2) board\_population = new\_population return None print(genetic\_algorithm(8))  
Источник: <https://www.easydoit.ru/python/reshenie-zadachi-o-8-ferzyax-s-pomoshhyu-python-bystro-prosto-i-elegantno/>

Полуработающий код

using System;

using System.Collections.Generic;

class Program

{

static Random random = new Random();

static List<int> GenerateBoard(int size)

{

List<int> board = new List<int>();

for (int i = 0; i < size; i++)

{

board.Add(random.Next(0, size));

}

return board;

}

static int Fitness(List<int> board)

{

int conflicts = 0;

for (int i = 0; i < board.Count; i++)

{

for (int j = i + 1; j < board.Count; j++)

{

if (board[i] == board[j] || board[i] + i == board[j] + j || board[i] - i == board[j] - j)

{

conflicts++;

}

}

}

return conflicts;

}

static Tuple<List<int>, List<int>> CrossOver(List<int> board1, List<int> board2)

{

int size = board1.Count;

int index = random.Next(1, size - 1);

List<int> child1 = new List<int>(board1.GetRange(0, index));

child1.AddRange(board2.GetRange(index, size - index));

List<int> child2 = new List<int>(board2.GetRange(0, index));

child2.AddRange(board1.GetRange(index, size - index));

return Tuple.Create(child1, child2);

}

static List<int> Mutate(List<int> board)

{

int size = board.Count;

int index = random.Next(0, size);

int newPos = random.Next(0, size);

board[index] = newPos;

return board;

}

static List<int> GeneticAlgorithm(int boardSize)

{

int populationSize = 10000;

int eliteSize = 2;

double mutationRate = 0.1;

List<List<int>> boardPopulation = new List<List<int>>();

for (int i = 0; i < populationSize; i++)

{

boardPopulation.Add(GenerateBoard(boardSize));

}

for (int i = 0; i < 1000; i++)

{

List<Tuple<int, List<int>>> fitnessScores = new List<Tuple<int, List<int>>>();

foreach (var board in boardPopulation)

{

fitnessScores.Add(Tuple.Create(Fitness(board), board));

}

fitnessScores.Sort((x, y) => x.Item1.CompareTo(y.Item1));

List<List<int>> bestBoards = new List<List<int>>();

for (int j = 0; j < eliteSize; j++)

{

bestBoards.Add(fitnessScores[j].Item2);

}

if (fitnessScores[0].Item1 == 0)

{

return fitnessScores[0].Item2;

}

List<List<int>> newPopulation = new List<List<int>>(bestBoards);

while (newPopulation.Count < populationSize)

{

List<int> parent1 = bestBoards[random.Next(0, eliteSize)];

List<int> parent2 = bestBoards[random.Next(0, eliteSize)];

var children = CrossOver(parent1, parent2);

if (random.NextDouble() < mutationRate)

{

children = Tuple.Create(Mutate(children.Item1), children.Item2);

}

if (random.NextDouble() < mutationRate)

{

children = Tuple.Create(children.Item1, Mutate(children.Item2));

}

newPopulation.Add(children.Item1);

newPopulation.Add(children.Item2);

}

boardPopulation = newPopulation;

}

return null;

}

static void Main(string[] args)

{

List<int> result = GeneticAlgorithm(8);

if (result != null)

{

Console.WriteLine(string.Join(", ", result));

}

else

{

Console.WriteLine("No solution found.");

}

}

}