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)];

}

}

}