**Experiment:5**

**Aim: Write a Program to Implement 8-Queens Problem.**

**Description of 8-Queens Problem:**

In the context of artificial intelligence (AI), the 8-Queens Problem is a classic example often used to demonstrate various problem-solving techniques and algorithms, especially those related to constraint satisfaction and search algorithms.

The problem involves placing eight queens on an 8x8 chessboard in such a way that no two queens threaten each other. A queen can attack another piece if it is on the same row, column, or diagonal as the other piece. Therefore, in the 8-Queens Problem, the objective is to find a configuration of queens where none of them can attack each other.

Several common approaches are employed to solve the 8-Queens Problem in AI. Each method employs different strategies and algorithms to find solutions:

1. **Brute Force:** Brute force involves systematically trying all possible arrangements of queens on the chessboard and checking each arrangement to see if it satisfies the constraints of the problem. While this method is straightforward, it becomes impractical for larger chessboards due to the exponential growth in the number of possible configurations.
2. **Backtracking:** Backtracking is a more efficient approach that involves systematically exploring the search space and backtracking when a dead-end is reached. The algorithm explores the solution space recursively, discarding partial solutions that violate the constraints. Backtracking is particularly well-suited for solving constraint satisfaction problems like the 8-Queens Problem.
3. **Constraint Satisfaction:** The 8-Queens Problem can be formulated as a constraint satisfaction problem (CSP), where the objective is to find a configuration of queens that satisfies all the constraints. Various CSP-solving techniques, such as constraint propagation, arc consistency, and constraint satisfaction algorithms like AC-3 or backtracking with constraint propagation (e.g., forward checking), can be applied to solve the problem efficiently.
4. **Genetic Algorithms:** Genetic algorithms (GAs) are inspired by the process of natural selection and evolution. In this approach, a population of potential solutions is evolved over successive generations through processes such as selection, crossover, and mutation. Genetic algorithms can be applied to the 8-Queens Problem by representing potential solutions as chromosomes and evolving them to find solutions that satisfy the constraints.
5. **Simulated Annealing:** Simulated annealing is a probabilistic optimization technique inspired by the process of annealing in metallurgy. In this approach, the algorithm starts with an initial solution and iteratively explores neighboring solutions, accepting solutions that improve the objective function and occasionally accepting worse solutions to escape local optima. Simulated annealing can be adapted to solve the 8-Queens Problem by defining an appropriate objective function and exploring neighboring configurations.
6. **Local Search Algorithms:** Local search algorithms, such as hill climbing and tabu search, iteratively explore the search space by making incremental changes to the current solution. These algorithms aim to find optimal or near-optimal solutions by iteratively improving upon the current solution. Local search algorithms can be adapted to solve the 8-Queens Problem by defining appropriate neighborhood structures and search heuristics.

**Genetic Algorithms (GAs)** are heuristic search and optimization techniques inspired by the principles of natural selection and genetics. They are commonly used to solve optimization and search problems where traditional algorithms may be impractical or inefficient.

The fundamental concept of Genetic Algorithms revolves around the idea of evolving a population of potential solutions over successive generations to find optimal or near-optimal solutions to a given problem.

Here's a basic description of how Genetic Algorithms work:

* **Initialization:** The algorithm starts by initializing a population of potential solutions to the problem randomly. Each potential solution is represented as a chromosome, typically a string or an array of values.
* **Evaluation:** The fitness of each individual in the population is evaluated using a fitness function. This function quantifies how well a solution performs with respect to the problem's objectives. The fitness function guides the search towards better solutions.
* **Selection:** Individuals from the current population are selected for reproduction based on their fitness. Higher-fitness individuals are more likely to be selected for reproduction, mimicking the principle of "survival of the fittest" in natural selection.
* **Crossover (Recombination):** During crossover, pairs of selected individuals (parents) exchange genetic information to produce offspring (children). This process is analogous to genetic recombination in biological organisms. Different crossover techniques, such as single-point crossover, multi-point crossover, or uniform crossover, can be used to create diverse offspring.
* **Mutation:** After crossover, random changes (mutations) are introduced into the offspring's genetic material with a certain probability. Mutation helps maintain diversity in the population and prevents premature convergence to suboptimal solutions.
* **Replacement:** The new offspring and some individuals from the current population are combined to form the next generation. The selection of individuals for the next generation can involve strategies like elitism (preserving the best individuals) or stochastic replacement.
* **Termination:** The algorithm iterates through the steps of selection, crossover, mutation, and replacement for a fixed number of generations or until a termination criterion is met (e.g., reaching a satisfactory solution, convergence).

**Implementation of 8 Queens Problem -:**

The eight queens problem is to arrange eight queens on an 8 by 8 chessboard so that no two are in the same row, column, or diagonal and hence cannot attack one another. Generally speaking, an n×n chessboard has n queens in the n queens problem. The method begins by inserting a queen into the first column. It then moves on to the second column and inserts a queen into the column's first safe row. The program prints the board and returns true if it reaches the eighth column and all of the queens are positioned safely. The algorithm goes back to the previous column and attempts a new row if it is unable to position a queen in a secure location in that column.

**Implementation of 8-Queens Problem using Genetic Algorithms**

import random

class EightQueensGA:

def \_\_init\_\_(self, population\_size, mutation\_rate):

self.population\_size = population\_size

self.mutation\_rate = mutation\_rate

self.population = []

def generate\_individual(self):

return [random.randint(0, 7) for \_ in range(8)]

def generate\_population(self):

self.population = [self.generate\_individual() for \_ in range(self.population\_size)]

def fitness(self, individual):

clashes = 0

for i in range(len(individual)):

for j in range(i + 1, len(individual)):

if individual[i] == individual[j] or abs(i - j) == abs(individual[i] - individual[j]):

clashes += 1

return clashes

def crossover(self, parent1, parent2):

crossover\_point = random.randint(0, 7)

child = parent1[:crossover\_point] + parent2[crossover\_point:]

return child

def mutate(self, individual):

if random.random() < self.mutation\_rate:

index = random.randint(0, 7)

individual[index] = random.randint(0, 7)

def evolve(self):

new\_population = []

# Elitism: Keep the best individual from the previous generation

best\_individual = min(self.population, key=self.fitness)

new\_population.append(best\_individual)

while len(new\_population) < self.population\_size:

parent1 = random.choice(self.population)

parent2 = random.choice(self.population)

child = self.crossover(parent1, parent2)

self.mutate(child)

new\_population.append(child)

self.population = new\_population

def solve(self, max\_generations):

self.generate\_population()

generation = 0

while generation < max\_generations:

self.evolve()

generation += 1

best\_individual = min(self.population, key=self.fitness)

if self.fitness(best\_individual) == 0:

print("Solution found in generation", generation)

print("Queens positions:", best\_individual)

return best\_individual

print("Solution not found after", max\_generations, "generations")

return None

if \_\_name\_\_ == "\_\_main\_\_":

population\_size = 100

mutation\_rate = 0.1

max\_generations = 1000

ga = EightQueensGA(population\_size, mutation\_rate)

solution = ga.solve(max\_generations)

This implementation uses a simple genetic algorithm to solve the 8-Queens Problem. The main steps include generating an initial population, calculating fitness for each individual, performing selection, crossover, mutation, and evolving the population over generations until a solution is found or the maximum number of generations is reached.

**Output:**

Solution found in generation 848

Queens positions: [4, 1, 5, 0, 6, 3, 7, 2]

**Date of experiment performed:**

**Day of experiment performed: Tuesday**

**Date of experiment submission:**

**Day of experiment Submission: Tuesday**

Faculty Co-ordinator Signature