Genetic Algorithm and the Eight Queens Problem **(Translated By Google Translate)**

**Introduction**

Genetic algorithms are a subset of evolutionary algorithms. This algorithm is inspired by the body biological system. The algorithm needs to be able to represent each possible answer to the problem in a suitable structure (this representation is called chromosome representation).

Next, a function needs to map each instance of the answer (displayed in the appropriate structure) to a numeric value. This function must indicate how appropriate an answer is to the problem. This function is called the fitness function.

After determining the structure of the representation of an answer in a problem and its evaluation function, an initial generation of responses is produced and by applying genetic processes, the initial generation is created from the next generations and this process is repeated to find the appropriate answer.

This algorithm is suitable for finding answers to problems that have a very wide search space and known numerical ways can not help us to find the answer. Also, in situations where we do not have a mathematical knowledge of the problem but can measure the appropriateness of an answer, using these methods can help find the answer to the problem.

The answers obtained from the genetic algorithm are not the best possible answer. In fact, there is no guarantee of finding the optimal answer using this algorithm. But using this algorithm, you can find answers that are sufficiently appropriate. As a result, it is not appropriate to use this method in problems that require an absolute optimal answer, but to find an answer close to the optimal answer can be used.

**Steps of this algorithm**

1. Production of the first generation Initialization

2. Selection

3. Genetic Operations

**Production of the first generation**

In this phase, a number of answers to the problem are generated randomly (usually between 50 and 100 answers).

**Selection**

From the previous generation, a number of answers are selected to generate new answers. Different methods have been suggested for this selection. This selection can be completely random. Another way can be used to have some of the best answers in our selection and other answers are selected randomly. Another suggestion is to choose randomly from the previous population, with the difference that the probability of selecting the answers that are considered more appropriate for the problem is higher than other answers (this method is also called roulette wheel).

**Genetic operators**

After selecting the next generation parents, we create the next generation using genetic operators. One point to note is that the number of responses per generation is considered equal and does not usually change the size of the generation.

Two popular operators to create a new generation are:

1. Crossover

2. Mutation

The Crossover operation is such that a combination of information from two parents creates a new response.

How to define this operator can be defined differently depending on the problem definition and how the answer is displayed.

The act of Mutation is that the response values ​​change randomly with one probability (usually this probability is low) and take on other values.

Termination conditions

After performing the above operation, a new generation of answers is obtained. The question is when the process should stop. There is no exact answer for the end condition. One suggestion is to look at the number of fixed generations. For example, repeat the above process for 1000 generations and choose the best answer from them. Another suggestion may be to repeat this process until you reach the desired fitness assessment value.

**Implementing the Eight Queen problem using a genetic algorithm**

**Define the problem**

Place eight queens on a chessboard so that they do not threaten each other.

This problem falls into the category of CSP (Constraint Satisfaction Problems) and can be solved in different ways. In the following, we will solve this problem to show how to use the genetic algorithm.

**Define the appropriate structure to answer this question**

There are different ways to display the answers to this question. In this paper, the problem is represented using a vector of length 8. The value of each element of this vector can be between 0 and 7. The interpretation of this structure is such that each element of the vector represents the location of Quinn in a column. For example, the third column specifies the location of the queen in the third column. This display method automatically eliminates the possibility of examining the answers that two queens place in a column.

**Define the evaluation function**

To evaluate each answer at the beginning, the number of threats to each queen is determined in pairs. This is a maximum of 28 because C (8, 2) = 28 is a choice of two out of eight times twenty-eight. This number is then divided by the maximum value, which is 28. The higher the value, the better the answer to the problem.

*fitness\_function(Entity &e) {*

*int threats = 0;*

*// total possible threats = C(2, 8) = 28*

*// this value is for 8 queens*

*const int total\_posible\_threats = 28;*

*for (int col = 0; col < Entity::COUNT\_COLUMN; col++) {*

*int col\_val = e.get\_column(col);*

*for (int ptr = col + 1; ptr < Entity::COUNT\_COLUMN; ptr++) {*

*int ptr\_val = e.get\_column(ptr);*

*if (col\_val == ptr\_val) {*

*// in same row*

*threats++;*

*} else if (ptr - col == ptr\_val - col\_val) {*

*// diagonal `/`*

*threats++;*

*} else if (ptr - col == col\_val - ptr\_val) {*

*// diagonal `\`*

*threats++;*

*}*

*}*

*}*

*return 1 - (threats / (float)total\_posible\_threats);*

*}*

First generation production

The first generation is created completely randomly. To create each answer, a random number between 0 and 7 is placed in each vector.

*default\_random\_engine Entity::generator;*

*uniform\_int\_distribution<int> Entity::distribution(0, COUNT\_ROW-1);*

*Entity\* Entity::generate\_random\_entity() {*

*Entity \*entity = new Entity();*

*for (int col = 0; col < COUNT\_COLUMN; col++) {*

*int row = distribution(generator);*

*entity->set\_column\_row(col, row); }*

*return entity; }*

*Entity\* generate\_random\_population(int size) {*

*Entity\* population =new Entity[size];*

*Entity\* e;*

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

*e = Entity::generate\_random\_entity();*

*population[i] = \*e;*

*}*

*return population;*

*}*

**Selection**

The roulette wheel method has been used to select from among a generation. In this method, the probability of selecting each answer is proportional to the evaluation value of that answer.

*Entity\* select\_from\_population(Entity\* population, const int size)*

*{ float comulative\_f[size];*

*float normal\_f;*

*float comulative = 0;*

*float total\_f = 0;*

*// calculate sum of f value for normalization*

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

*total\_f += fittness\_function(population[i]);*

*// generate the commulative value*

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

*normal\_f = fittness\_function(population[i]) / total\_f;*

*comulative += normal\_f;*

*comulative\_f[i] = comulative;*

*}*

*// select parents randomly with respect to fittness*

*Entity\* selected = new Entity[size];*

*double max\_rnd = (double)(RAND\_MAX) + 1;*

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

*double rnd = rand() / max\_rnd;*

*int ptr = 0;*

*while (ptr < size - 1) {*

*if (comulative\_f[ptr] >= rnd)*

*break; ptr++;*

*}*

*selected[i] = population[ptr];*

*}*

*return selected;*

*}*

**Application of genetic operators**

In this section, the cut is applied to the selected answers. This section is defined in such a way that first the answers are divided into two groups and then a point of the answers is randomly selected and the information of the two answers is moved before and after that point.

*Entity\* cross\_over(Entity\* population, const int size) {*

*if (size % 2 != 0)*

*throw invalid\_argument("population size is not even number!");*

*Entity\* result = new Entity[size / 2];*

*int pop\_index = 0;*

*for (int i = 0; i < size; i += 2) {*

*Entity\* xover = new Entity();*

*int xover\_pnt = rand() % Entity::COUNT\_COLUMN;*

*for (int j = 0; j < Entity::COUNT\_COLUMN; j++) {*

*int val;*

*if (j <= xover\_pnt) {*

*val = population[i].get\_column(j);*

*}*

*else {*

*val = population[i + 1].get\_column(j);*

*}*

*xover->set\_column\_row(j, val);*

*}*

*result[pop\_index++] = \*xover;*

*}*

*return result;*

*}*

**End condition**

The cycle of creating a new generation is repeated until an answer with an estimated value of one is found.