Definition

- **Genetic Algorithms is a search method in which multiple search paths are followed in parallel. At each step, current states of different pairs of these paths are combined to form new paths. This way the search paths don't remain independent, instead they share information and thus try to improve the overall performance of the complete search space**

The Basic Genetic Algorithm

- **Start with a population of randomly generated attempted solutions to a problem**
- **Repeatedly do the following:**
  - Evaluate each of the attempted solutions
  - Keep the “best” solutions
  - Produce next generation from these solutions (using “inheritance” and mutation)
- **Quit when you have a satisfactory solution (or you run out of time)**

A Simple Example

- Suppose your "individuals" are 32-bit computer words
- You want a string in which all the bits are ones
- Here's how you can do it:
  - Create 100 randomly generated computer words
  - Repeatedly do the following:
    - Count the 1 bits in each word
    - Exit if any of the words have all 32 bits set to 1
    - Keep the ten words that have the most 1s (discard the rest)
    - From each word, generate 9 new words as follows:
      - Pick a random bit in the word and toggle (change) it
- Note that this procedure does not guarantee that the next "generation" will have more 1 bits, but it's likely

Initial Population

Evaluation Function

Mutation

A More Realistic Example

- Suppose you have a large number of (x, y) data points
  - For example, (1, 4), (3, 9), (5, 8), …
- You would like to fit a polynomial (of up to degree 1) through these data points
  - That is, you want a formula y = mx + c that gives you a reasonably good fit to the actual data
  - Here's the usual way to compute goodness of fit:
    - Compute the sum of (actual y – predicted y)2 for all the data points
    - The lowest sum represents the best fit
- You can use a genetic algorithm to find a "pretty good" solution

A More Realistic Example

- Your formula is y = mx + c
- Your unknowns are m and c; where m and c are integers
- Your representation is the array [m, c]
- Your evaluation function for one array is:
  - For every actual data point (x, y)
    - Compute ý = mx + c
    - Find the sum of (y – ý)2 over all x
    - The sum is your measure of "badness" (larger numbers are worse)
  - Example: For [m,c] = [5, 7] and the data points (1, 10) and (2, 13):
    - ý = 5x + 7 = 12 when x is 1
    - ý = 5x + 7 = 17 when x is 2
    - Now compute the badness
      (10 - 12)2 + (13 – 17)2 = 22 + 42 = 20
    - **If these are the only two data points, the "badness" of [5, 7] is 20**

A More Realistic Example

- Your GA might be as follows:
  - Create two-element arrays of random numbers
  - Repeat 50 times (or any other number):
    - For each of the arrays, compute its badness (using all data points)
    - Keep the best arrays (with low badness)
    - From the arrays you keep, generate new arrays as follows:
      - Convert the numbers in the array to binary, toggle one of the bits at random
  - Quit if the badness of any of the solution is zero
  - After all 50 trials, pick the best array as your final answer

A More Realistic Example

- (x, y) : {(1,5) (3, 9)}
- [2 7][1 3] (initial random population, where m and c represent genes)
  - ý = 2x + 7 = 9 when x is 1
  - ý = 2x + 7 = 13 when x is 3
  - Badness: $(5 – 9)^2 + (9 – 13)^2 = 4^2 + 4^2 = 32$
  - ý = 1x + 3 = 4 when x is 1
  - ý = 1x + 3 = 6 when x is 3
  - Badness: $(5 – 4)^2 + (9 – 6)2 = 1^2 + 3^2 = 10$
- Now, lets keep the one with low "badness" [1 3]
- Binary representation [001 011]
- Apply mutation to generate new arrays [011 011]
- Now we have [1 3] [3 3] as the new population considering that we keep the two best individuals

A More Realistic Example

- (x, y) : {(1,5) (3, 9)}
- [1 3][3 3] (current population)
  - ý = 1x + 3 = 4 when x is 1
  - ý = 1x + 3 = 6 when x is 3
  - Badness: $(5 - 4)^2 + (9 - 6)^2 = 1 + 9 = 10$
  - ý = 3x + 3 = 6 when x is 1
  - ý = 3x + 3 = 12 when x is 3
  - Badness: $(5 - 6)^2 + (9 - 12)^2 = 1 + 9 = 10$
- Lets keep the [3 3]
- Representation [011 011]
- Apply mutation to generate new arrays [010 011] i.e. [2,3]
- Now we have [3 3] [2 3] as the new population

A More Realistic Example

- (x, y) : {(1,5) (3, 9)}
- [3 3][2 3] (current population)
  - ý = 3x + 3 = 6 when x is 1
  - ý = 3x + 3 = 12 when x is 3
  - Badness: $(5 - 6)^2 + (9 - 12)^2 = 1 + 9 = 10$
  - ý = 2x + 3 = 5 when x is 1
  - ý = 2x + 3 = 9 when x is 3
  - Badness: $(5 - 5)^2 + (9 - 9)^2 = 0^2 + 0^2 = 0$
- Solution found [2 3]
- y = 2x+3
- Note: It is not necessary that the badness must always be zero. It can be some other threshold value as well.

The simple example again

- Suppose your "individuals" are 32-bit computer words, and you want a string in which all the bits are ones
- Here's how you can do it:
  - Create 100 randomly generated computer words
  - Repeatedly do the following:
    - Count the 1 bits in each word
    - Exit if any of the words have all 32 bits set to 1
    - Keep the 10 words that have the most 1s (discard the rest).
    - From each word, generate 9 new words as follows:
      - Choose one of the words
      - Take the first half of this word and combine it with the second half of some other word

The simple example again

- Half from one, half from the other:

A = 0110 1001 0100 1110 1010 1101 1011 0101

B = 1101 0100 0101 1010 1011 0100 1010 0101

C = 0110 1001 0100 1110 1011 0100 1010 0101

Mutation vs Crossover

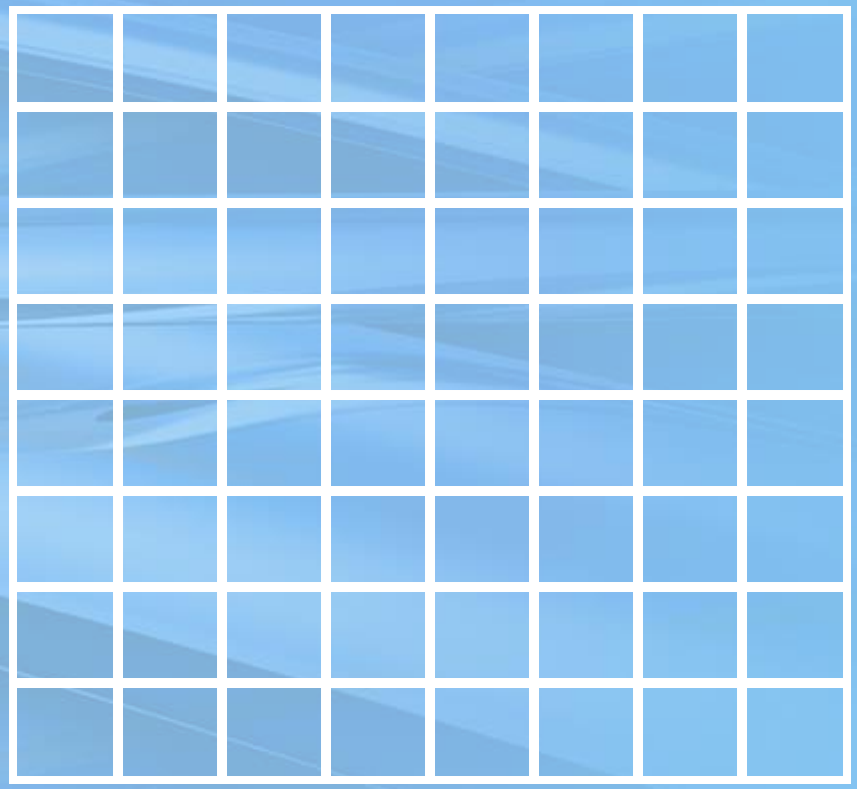
- In the simple example of 32-bit words (trying to get all 1s):
  - The (two-parent, no mutation) approach, if it succeeds, is likely to succeed much faster
    - Because up to half of the bits change each time, not just one bit
  - However, without mutation, it may not succeed at all
    - By pure bad luck, maybe none of the first randomly generated words have (say) bit 17 set to 1
      - Then there is no way a 1 could ever occur in this position as we are not changing individual bits separately
    - Another problem is lack of genetic diversity
      - Maybe some of the first generation did have bit 17 set to 1, but none of them were selected for the second generation
- The best technique in general turns out to be crossover with mutation

Assignment Curve Fitting

- Your formula is $y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$
- Your "genes" are a, b, c, d, e, and f
- Your "solution"/chromosome is the array [a, b, c, d, e, f]
- Generate a random initial population of likely solutions to the problem
- Apply Genetic Algorithm using Crossover and Mutation both to find the best curve that fits a given data set of (x, y) points

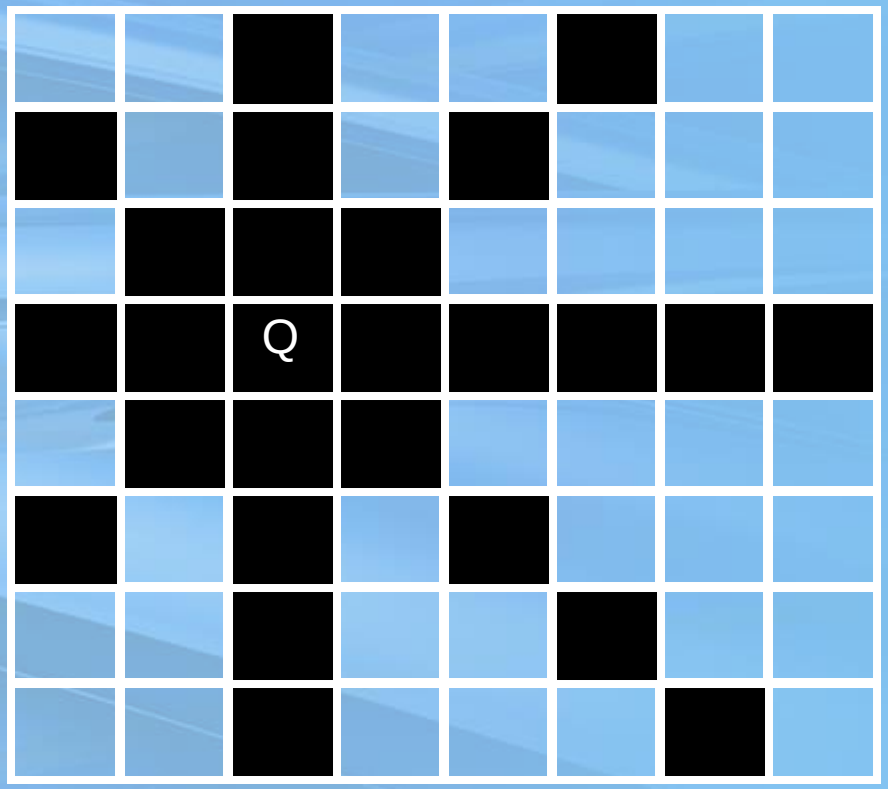
Eight Queens Problem

- **The problem is to place 8 queens on a chess board so that none of them can attack the other. A chess board can be considered a plain board with eight columns and eight rows.**



Eight Queens Problem

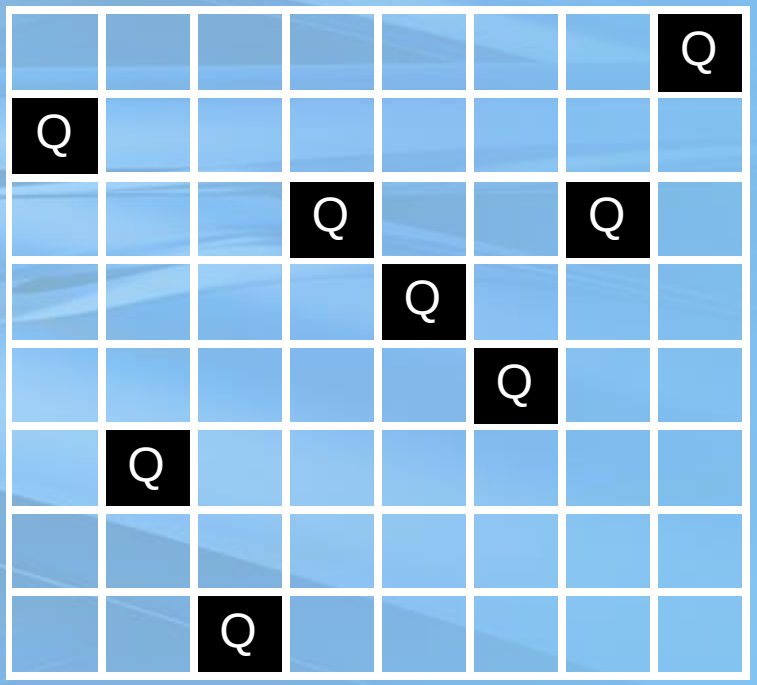
- The possible cells that the Queen can move to when placed in a particular square are shaded



Eight Queens Problem

- We need a scheme to denote the board's position at any given time

2 6 8 3 4 5 3 1

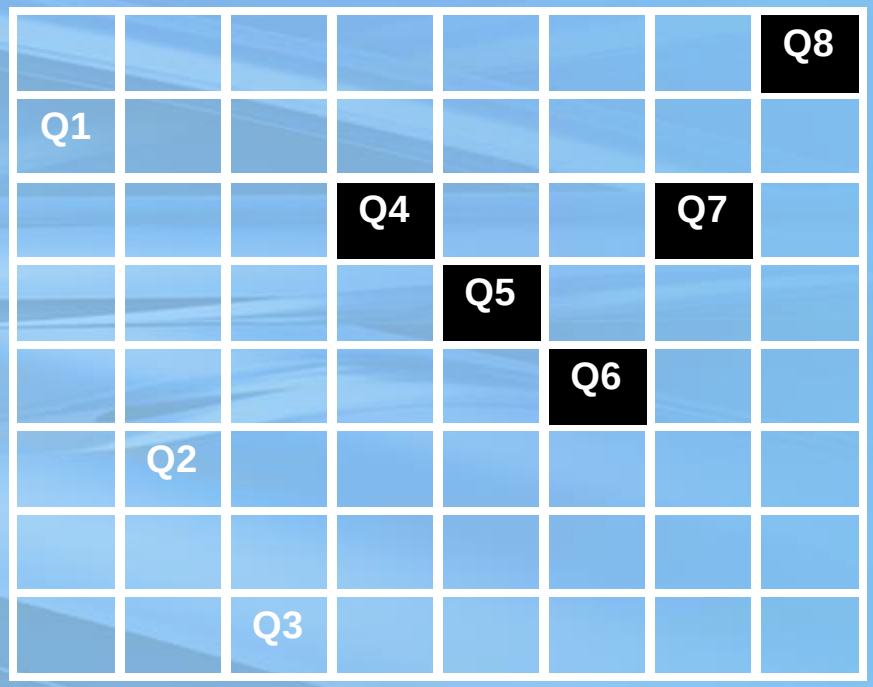


Eight Queens Problem

- Now we need a fitness function, a function by which we can tell which board position is nearer to our goal. Since we are going to select best individuals at every step, we need to define a method to rate these board positions.
- One fitness function can be to count the number of Queens that do not attack others

Eight Queens Problem

Fitness Function:
Q1 can attack NONE
Q2 can attack NONE
Q3 can attack Q6
Q4 can attack Q5
Q5 can attack Q4
Q6 can attack Q5
Q7 can attack Q4
Q8 can attack Q5



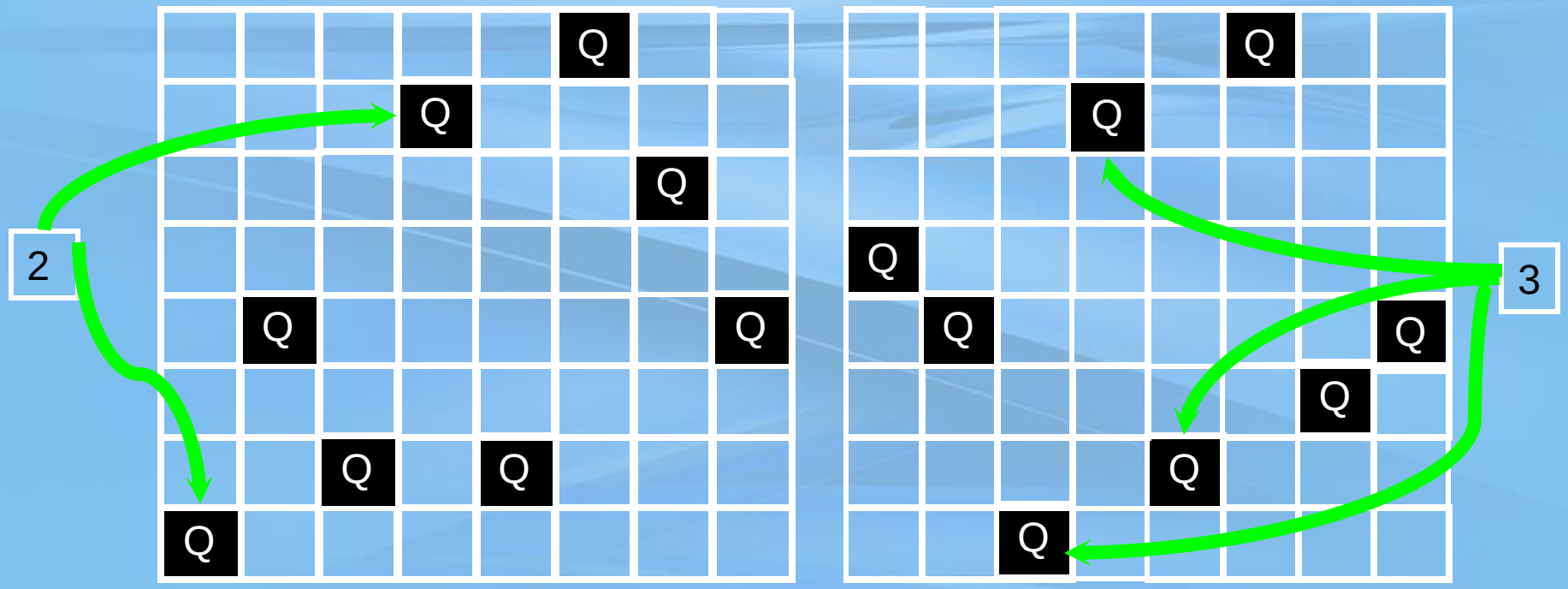
Fitness = No of. Queens that can attack none
Fitness = 2

Eight Queens Problem

- Choose initial population of board configurations
- Evaluate the fitness of each individual (configuration)
- Choose the best individuals from the population for crossover

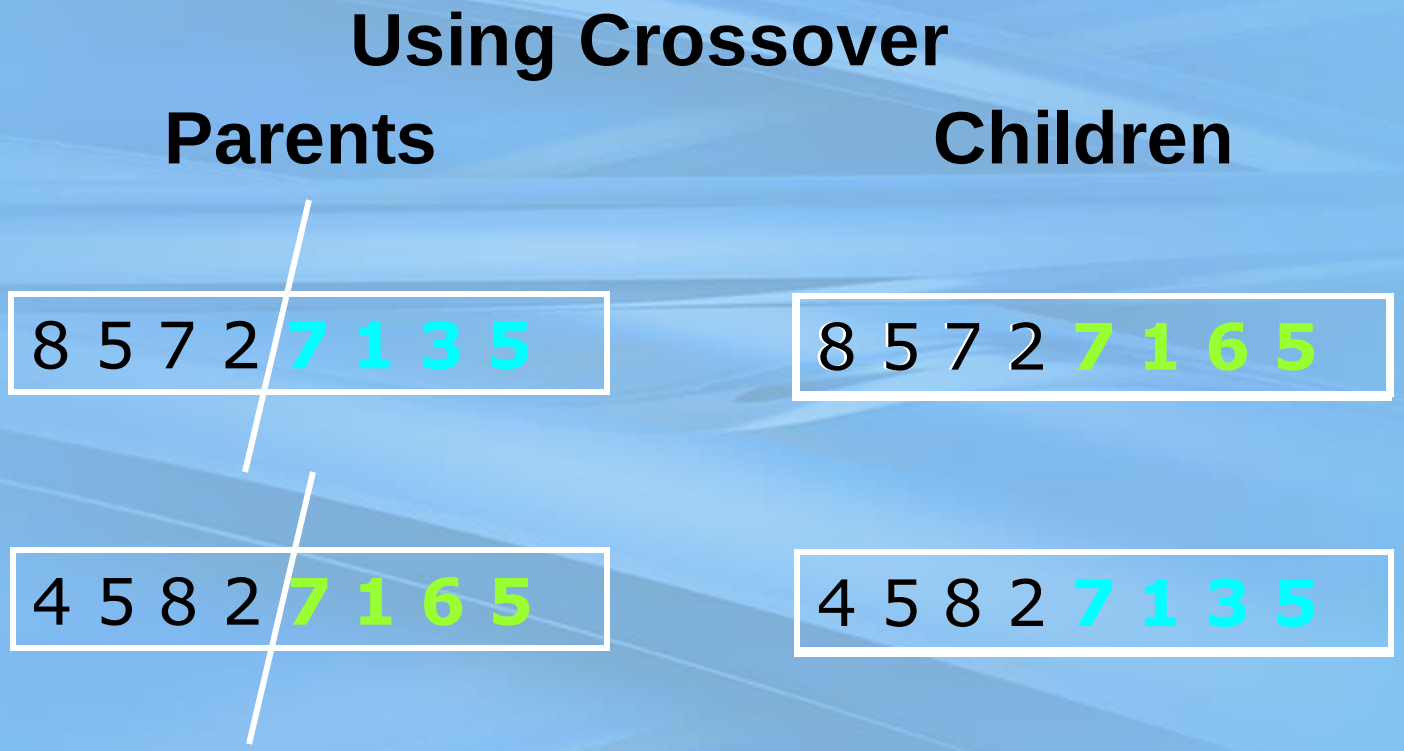
Eight Queens Problem

- **Suppose the following individuals are chosen for crossover**

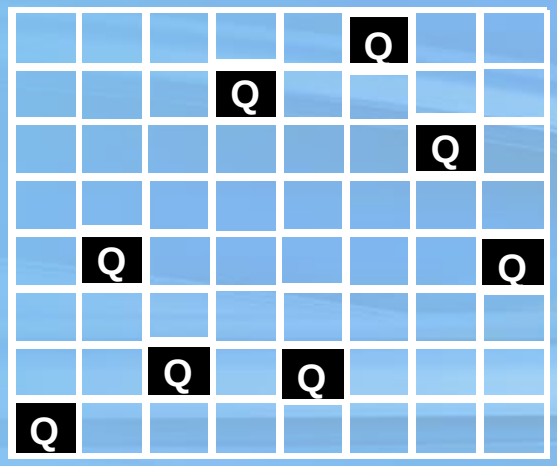


Eight Queens Problem

Using Crossover

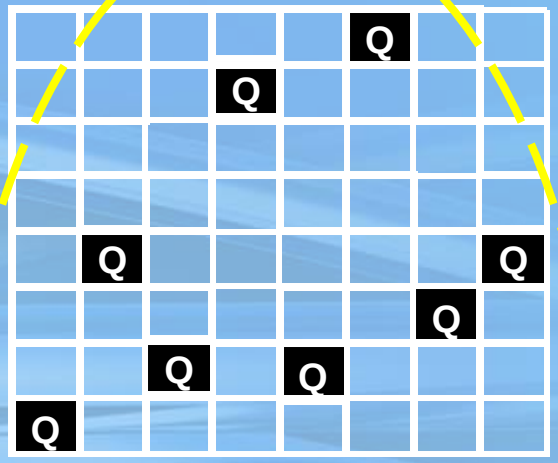


Eight Queens Problem



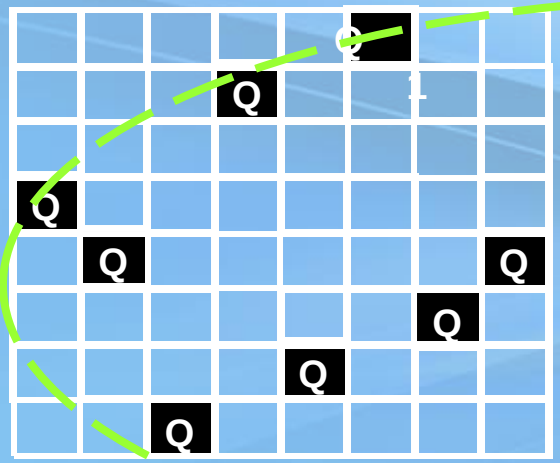
2

8 5 7 2 7 1 3 5



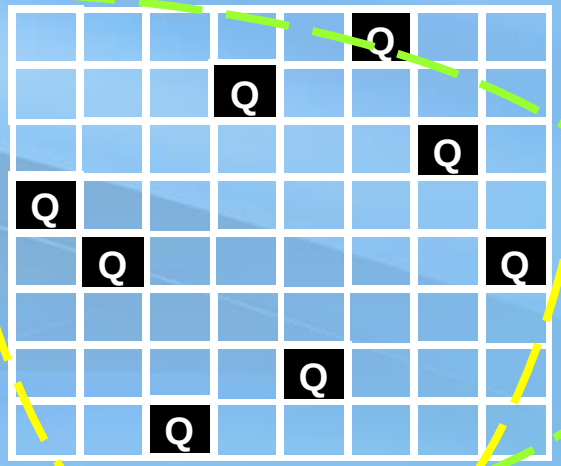
2

8 5 7 2 7 1 6 5



3

4 5 8 2 7 1 6 5



4

4 5 8 2 7 1 3 5

Eight Queens Problem

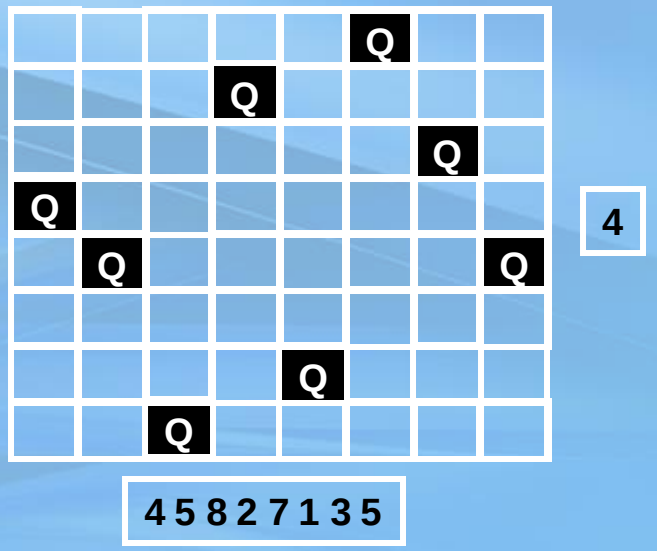
- **Mutation, flip bits at random**

4 5 8 2 7 1 6 5

0100 0101 1000 0010 0111 0001 0110 0101

0100 0101 1000 0010 0111 0001 0011 0101

4 5 8 2 7 1 3 5

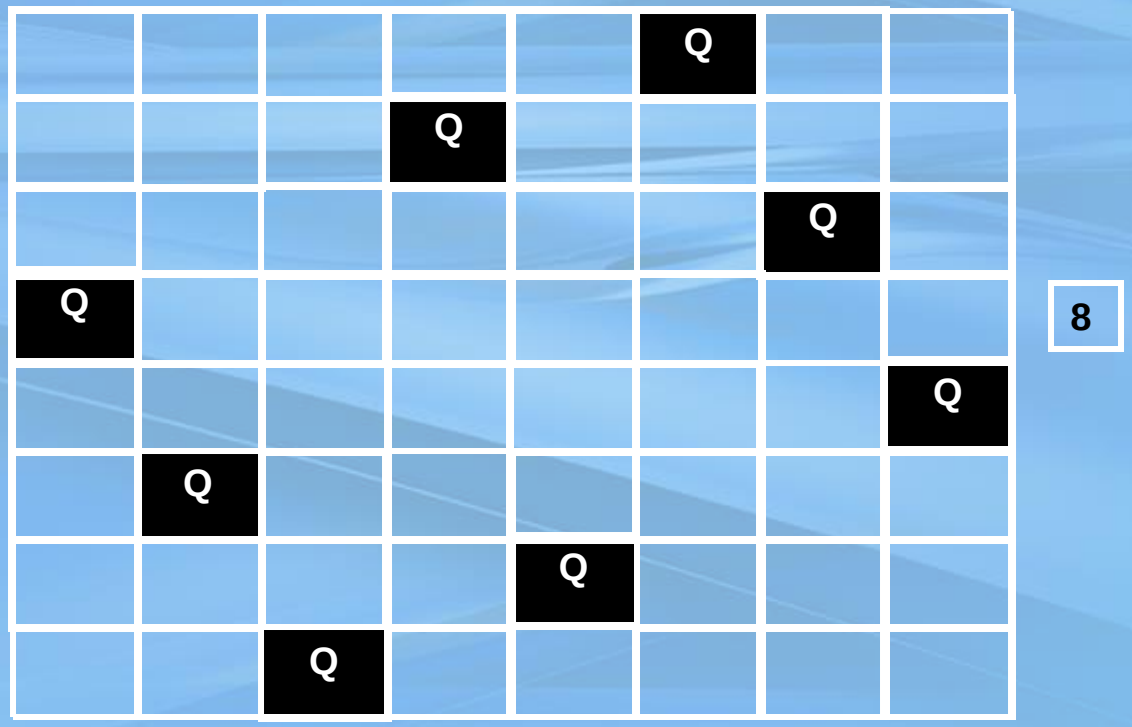


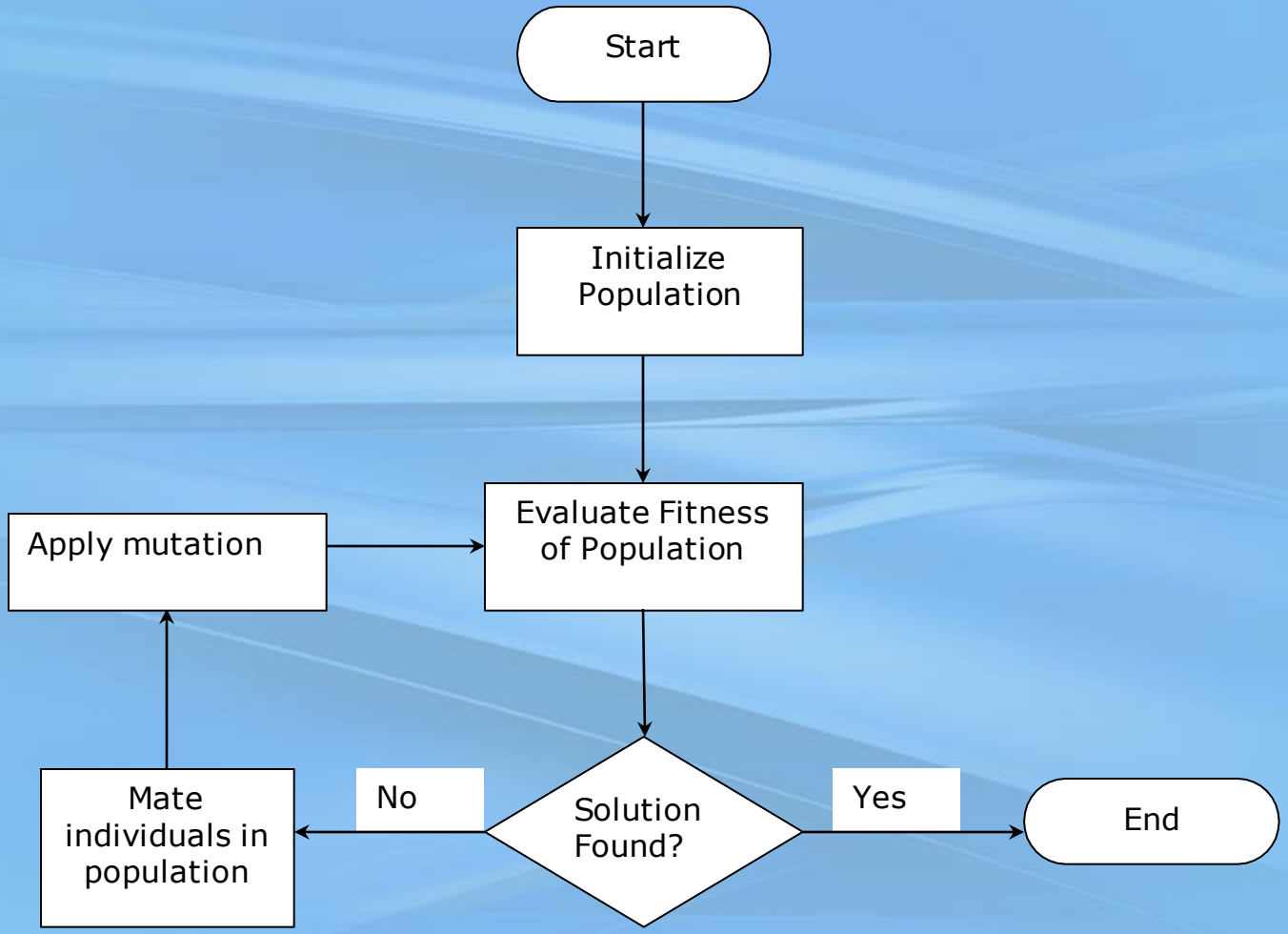
Eight Queens Problem

- **This process is repeated until an individual with required fitness level is found. If no such individual is found, then the process is repeated further until the overall fitness of the population or any of its individuals gets very close to the required fitness level. An upper limit on the number of iterations is usually put to end the process in finite time.**

Eight Queens Problem

- **Solution!**




Start
Initialize Population
Evaluate Fitness of Population
Apply mutation
Mate individuals in population
No
Solution Found?
Yes
End