Search Algorithms: Object-Oriented Implementation (Part F)

Contents

유전 알고리즘

GENETIC ALGORITHM (GA)

Genetic Algorithm

- Genetic Algorithm, GA(유전 알고리즘)
  - A **computational model** based on the **evolutionary process** and **genetic laws** of the natural world, developed by **John Holland** in **1975** as a **Global Optimization Method**.
  - It is a representative technique of evolutionary computation that mimics the evolution of living organisms, borrowing many elements from the actual process of evolution. Concepts like **mutation, crossover (recombination) operation, generation, and population** are used in the problem-solving process.
  - When a problem is too complex to be computationally feasible (NP-hard, etc.), the Genetic Algorithm can be used to obtain a solution close to the optimal solution, even if the true optimal solution cannot be found.

https://ko.wikipedia.org/wiki/ 유전 _ 알고리즘

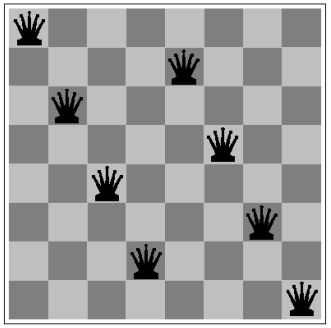
Genetic Algorithm

- Starts with a population of individuals (다수의 개별 솔루션으로 시작)
  - Each individual (state= 솔루션) is represented as a string over a finite alphabet (called chromosome, 염색체)—most commonly, a string of 0s and 1s (binary)

예 : 8 queens 문제에서의 하나의 솔루션 (state)

1	3	5	7	2	4	6	8

Column-by-column integer representation
(솔루션 = 각 열에서 queen 의 위치)



  - The method predominantly used to represent each Individual (state = solution) is **a string of 0s and 1s (binary)** (to be explained later).

Genetic Algorithm

- Each individual is rated by the fitness function
  - Solutions are selected based on the **fitness value**, and the selected solutions are used to generate **offspring** (offspring = next solution, offspring generation = reproduction).
  - An individual is selected for reproduction by the probability proportional to the fitness score

(참고) Local search algorithms that do not use any problem-specific heuristics is called "metaheuristic algo."

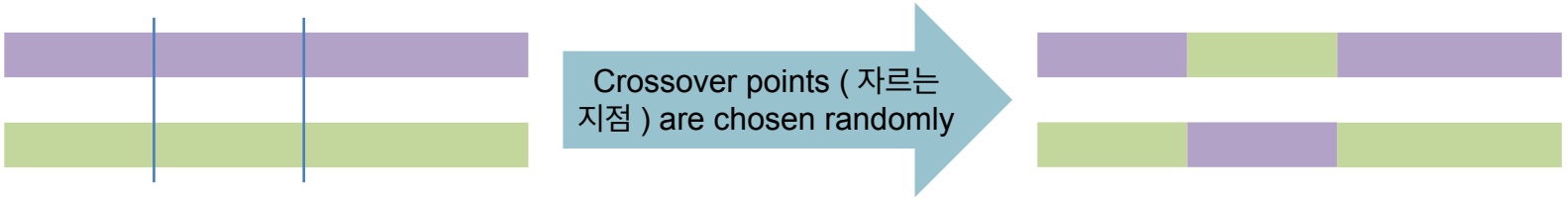
Simulated annealing and GA use meta-level heuristic → metaheuristic algorithms

Thereby,

- **Steepest Ascent, First-Choice, Stochastic, Gradient Descent**, etc., use **heuristic** techniques that are **specialized for a given problem**. (They are effective in solving the given problem, but cannot be applied to other problems.)
- **Simulated Annealing** and **GA** propose **heuristic** techniques using general ideas (such as annealing in daily life or genetic laws in nature) that are **not specific to a given problem**. (They do not change depending on the problem to be solved and can be applied to any problem.)

Genetic Algorithm

- Selected pair are mated (짝짓기) by a crossover (교차 연산)
  - After selecting a pair of chromosomes (=solutions), a **new solution** is generated using the **'crossover operation'** method, which involves randomly cutting the two chromosomes and splicing the resulting parts crosswise.
  - In this process, the **crossover point** (the cutting point) is determined randomly and can be a single point or multiple points.



  - Crossover frequently happens in the state space early in the search process when the population is quite diverse, and less frequently later on when most individuals are quite similar (다음 페이지…)

    ▸ This is similar to the principle used in algorithms like **Gradient Descent** and **Simulated Annealing**, where early in the learning process, the solutions are changed significantly (or the probability of selecting a poor solution is increased), and later in the learning process, the degree to which solutions are updated is reduced.

Genetic Algorithm

- Selected pair are mated (짝짓기) by a crossover (교차 연산)

Here's an example of how the probability of crossover can be incorporated into the crossover process:

```
def one_point_crossover(parent1, parent2, crossover_rate=0.8):
    if random.random() < crossover_rate:
        length = len(parent1)
        crossover_point = random.randint(1, length - 1)

        offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
        offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
    else:
        offspring1 = parent1.copy()
        offspring2 = parent2.copy()

    return offspring1, offspring2
```

* Note
In the lecture, we will implement Crossover using **different logic** from the code shown here. Please review the code provided here **for reference only**.

In this updated implementation, we introduce a `crossover_rate` parameter with a default value of 0.8. Before performing crossover, we generate a random number between 0 and 1 using `random.random()`. If this random number is less than the crossover rate, crossover is applied as before. Otherwise, the offspring solutions are direct copies of the selected parents.

The choice of the crossover rate can impact the balance between exploration and exploitation in the genetic algorithm. A higher crossover rate promotes more exploration by creating new offspring solutions through recombination. On the other hand, a lower crossover rate favors exploitation by preserving more of the genetic material from the selected parents.

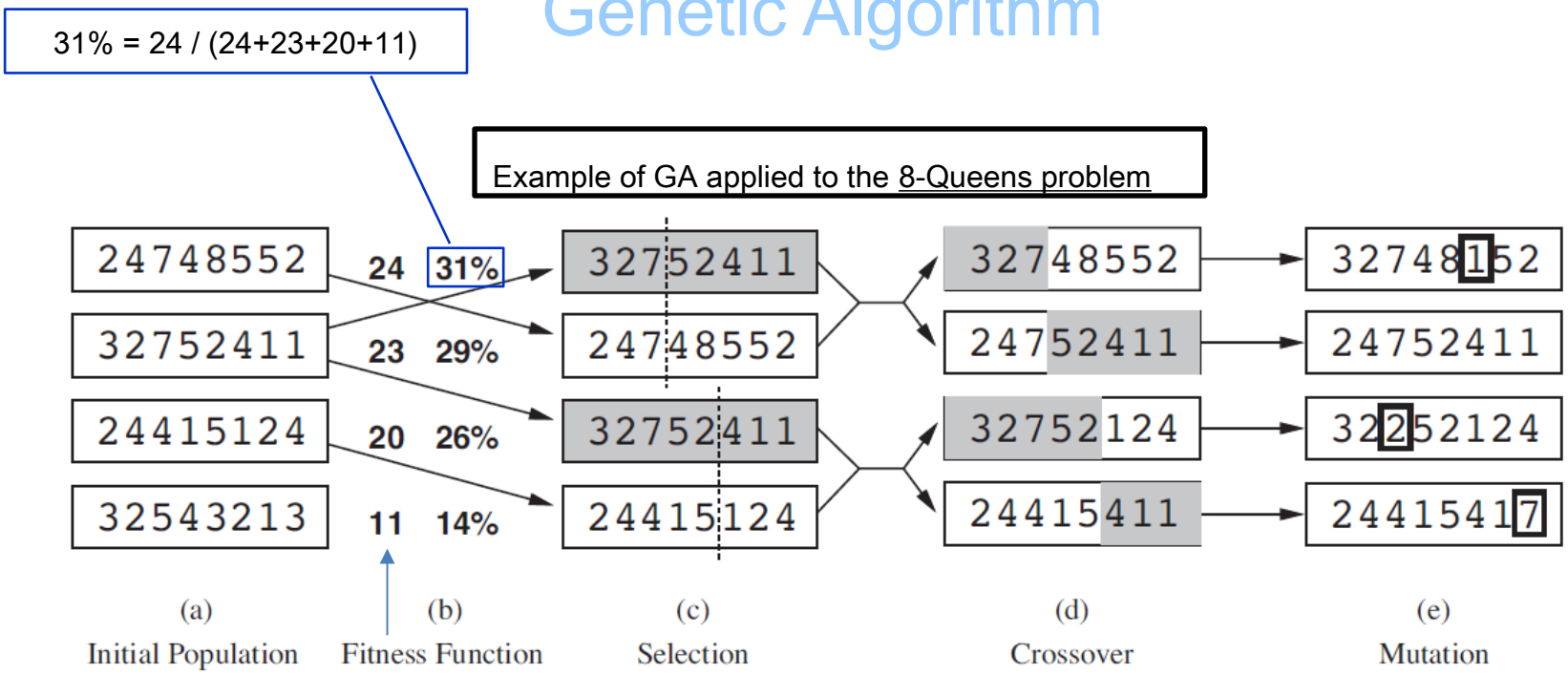
https://algorithmafternoon.com/books/genetic_algorithm/chapter05/

Genetic Algorithm

- Mutation Operation:
  - The **mutation operation** is applied after Crossover.
  - Each position (locus) in the **chromosome** (solution) is randomly changed with a small probability.
    - Each locus(위치 , 장소) is subject to random mutation with a small independent probability

- Advantages of the GA Technique :
  - crossover:
    - A method that combines two solutions that have been repeatedly developed/improved.
    - This significantly improves the **solution quality** (which is the main characteristic and the biggest advantage of the GA technique).
  - Mutation:
    - **Random variations** are applied to the **new solutions** created by Crossover.
    - The Mutation operation applies random changes to a single solution through **'randomness'** to search for better solutions.

Genetic Algorithm



(a) Starts with a diverse set of randomly selected solutions (**initial population**).

(b) Calculates the **fitness score** for the selected solutions.

(c) Each solution is **selected** with a probability proportional to its fitness function value.

(d) Two solutions are selected, and **crossover** is performed based on a random point to combine the two solutions and generate a new solution.

(e) Mutation (random change) is applied to each generated solution at a random position. However, since this process is probabilistic, there are cases where mutation is not applied.

Genetic Algorithm

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual

inputs: *population*, a set of individuals — an initial set of randomly generated solutions

FITNESS-FN, a function that measures the fitness of an individual

repeat

new_population ← empty set — Create an empty set to store the next generation's solutions

for $i = 1$ **to** SIZE(*population*) **do** — Generate new solutions until "new solutions = current population size."

(x, y) ← SELECT-PARENTS(*population*, FITNESS-FN) — Select two solutions by fitness scores

child ← REPRODUCE(x, y) — Create a new solution (**child**) by crossing over the two parent solutions.

if (small random probability) **then** *child* ← MUTATE(*child*) — Randomly change the solution

add *child* to *new_population* — Add the newly generated solution to **new_population**.

population ← *new_population* — **population = new_population**

until some individual is fit enough, or enough time has elapsed

return the best individual in *population*, according to FITNESS-FN — **best solution**

Genetic Algorithm

- Parent selection by binary tournament:

Binary

1. **Randomly select two (2) individuals with replacement** (i.e., the same solution can be selected twice) **without considering their fitness score**

Tournament

2. Select the one with the best fitness as the winner (ties are broken randomly)

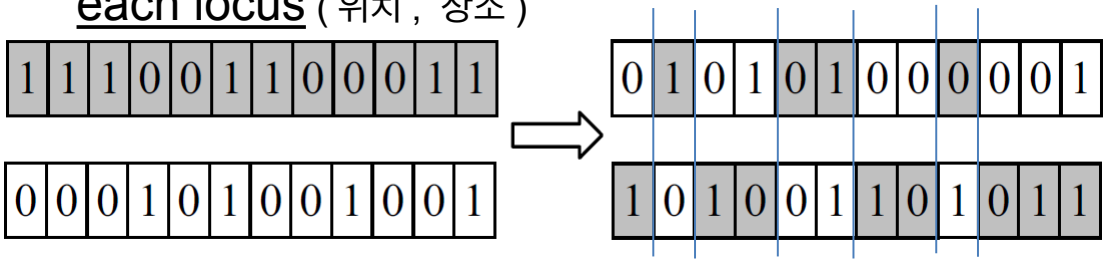
- **Repeat the above process twice** to select **two parents** from the current set of solutions.
- Use the two selected parent solutions to generate **one child (=next solution)**.

Genetic Algorithm

- Uniform crossover:

Uniform crossover is performed on **two parents** selected through the binary tournament.

  - Each gene is chosen from either parent stochastically by flipping coin at each locus (위치 , 장소)



Whether to perform **crossover** at each position is determined randomly (The image on the left shows an example where 6 crossover points are selected).

The crossover operation is performed to generate **two children** from **two parents**.

  - If the probability of head $p = 0.5$, the average number of crossover points is $l/2$, where l is the length of the chromosome (p=0.5 이면 평균적으로 절반이 교차됨)
  - $p = 0.2$ is a popular choice

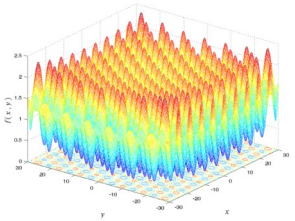
- After parent selection, crossover is carried out according to the probability called crossover rate
  - This is the probability that determines whether the **crossover operation itself** is performed. It is generally recommended to use a probability close to **1** (or a value that dynamically decreases over time may be used).

Genetic Algorithm

- Bit-flip mutation for binary representation:
  - The mutation operation is applied with a **low probability** to a single **child solution** generated through crossover.
  - Each bit is flipped with a small mutation probability called mutation rate
  - Widely-used mutation rate : $1/l$ (l = the length of one solution)

- However,
  - In order to perform operations like **Crossover** and **Mutation**, each solution must be represented in the form of an **array, list, or string**. (The GA technique itself originated from the idea of combining DNA sequences.)
  - The GA technique is a **Meta-Heuristic** method and must be applicable to all types of problems.
  - For example, when applying it to an optimization problem that determines x to minimize $x^4 - 2x^2$:,
    - With x=1.23 and x=3.42 how do you do crossover or mutation…?
    - Let's do BINARY ENCODING/DECODING

Genetic Algorithm



2-D Griewank function

Example: Binary encoding/decoding for numerical optimization

$$\min f(x,y) = \frac{x^2+y^2}{4000} - \cos(x)\cos\left(\frac{y}{\sqrt{2}}\right) + 1 \quad (-30 \le x, y \le 30)$$

– Assuming a 10-bit binary encoding for each variable, the code shown below can be decoded as (= discretization)

min

max

$$-30 + \left(30 - (-30)\right) \times \frac{1}{2^{10}} \left(2^8 + 2^4 + 2^0\right)$$

$$= -14.004$$

Dividing by 2^10 for 10-bit rep.

x =

0	1	0	0	0	1	0	0	0	1

Above: Example of generating one arbitrary solution (**assuming the solution range** is -30 to +30)
[1] Generate an arbitrary combination of **[10 bits]** and divide it by 2^10 (the result is a real number between 0 and 1).
[2] Multiply the **[calculated real number]** by the total range (=60) to scale the number to be between 0 and 60.
[3] Add -30 to **shift** the value into the valid range.

- In case of 10-bit, the maximum representation value is 2^{10}-1.
- therefore, representing 10-bit by dividing by 2^10, it becomes a real number between (0,1).
- This value is **scaled** and **shifted** to convert it into a number within the valid range.
- Note: as *n* in *n*-bit representation increases, the accuracy of the solution and computational load also increased

- Loss of information occurs due to **Discretization**, which can lead to **solution quality degradation**.
- However, it can reduce computational and memory usage.
- (Example) Deep learning **Model Optimization** - **Quantization**.

Genetic Algorithm

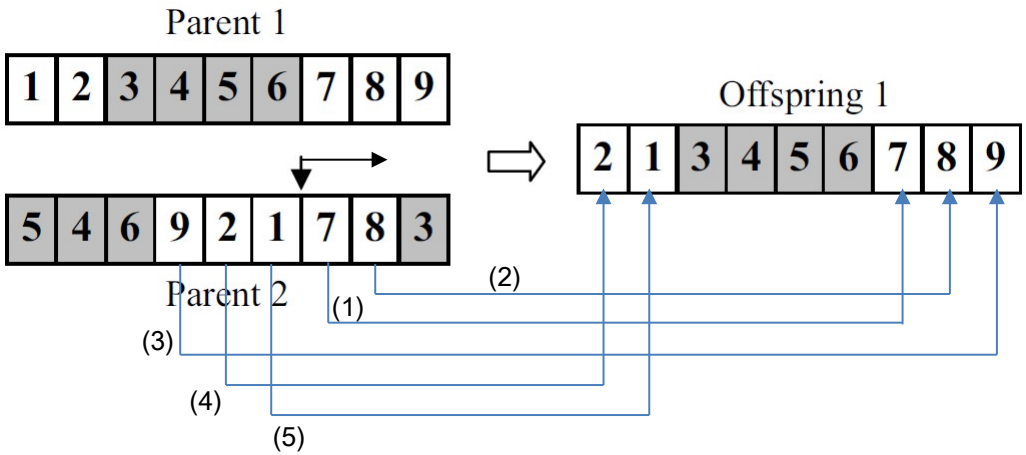
- Example for combinatorial problem (TSP)
  - Genetic operators for permutation code (e.g., for TSP):
    - The method used to represent information like the **order of visits** is called a **permutation code**.
    - Special caution is required in these types of problems.
    - This is because **invalid solutions** can be generated during the operations like crossover and mutation.
    - Ex:
      - ✓ [1,2,3,4] + [4,3,2,1] => [1,2,2,1]
      - ✓ The original two solutions are valid, but the solution created by crossover is **invalid** (it **does not visit all cities**).

Genetic Algorithm

The method used to represent information like the **order of visits**.

- Example for combinatorial problem (TSP)
  - Genetic operators for permutation code (e.g., for TSP):
    - How to generate **mutation** from a single solution: : Simple inversion mutation

    - How to cross over and combine two solutions: **Ordered** crossover (OX)
      - ✓ Ex: Parent 1's 3-4-5-6 is preserved/maintained



- If two solutions are simply combined based on a **Crossover point**, **certain city indices may disappear or duplicate visits may occur**.
- Left image operation example :
- Select a start point and an end point from **Parent 1**, and **copy the solution within that range as is** => [?, ?, 3, 4, 5, 6, ?, ?, ?]
- From **Parent 2**, starting **after the end index of Parent 1**, **copy the remaining values in order**, excluding values **already included in the offspring**, to create the new solution.

Genetic Algorithm

- In the case of permutation code
  - Crossover rate is the probability of whether or not to perform the ordered crossover
    - The probability that determines whether or not to perform the crossover operation using two solutions.
  - Similarly, mutation rate is the probability of whether or not to perform the inversion
    - The probability that determines whether or not to perform the mutation operation on a single solution.
    - Note that mutation in TSP = **mutate by inversion**.

GENETIC ALGORITHM (GA) IMPLEMENTATION

Adding Genetic Algorithm

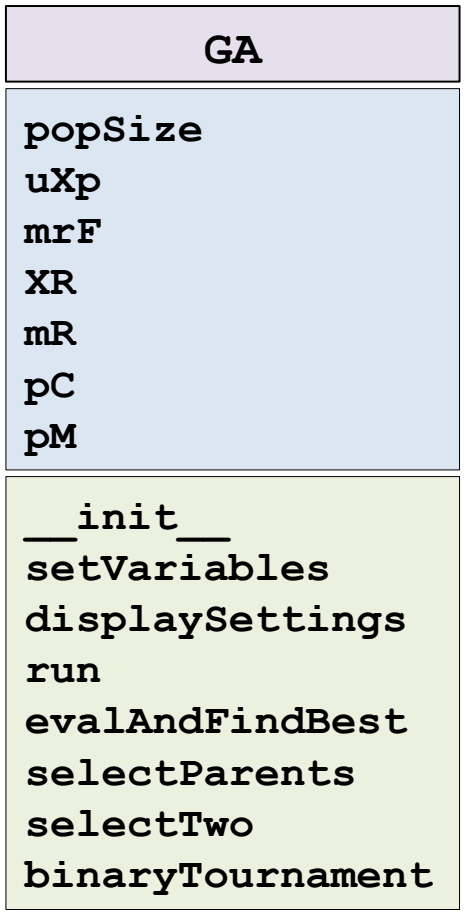
- We add a subclass **GA** under **MetaHeuristics** in the **Optimizer** class hierarchy
  - Accordingly, many problem-dependent methods for **GA** are added to the **Problem** class

- **GA** has seven variables:
  - **popSize**: population size (size of the solution set)
  - **uXp**: swap probability for uniform crossover used in numerical optimization (determine whether or not to cross over at each position of a solution of length l)
  - **mrF**: multiplication factor to $(1/l)$ for bit-flip mutation used in numerical optimization (A value used to determine the mutation probability)

numerical opt.

  - **XR**: crossover rate for the permutation code for TSP (The probability that determines whether or not to perform the crossover operation)

TSP



Adding Genetic Algorithm

- **mR**: mutation rate for the permutation code for solving TSP
- **pC**: crossover probability (**uXp** or **XR**)
- **pM**: mutation probability (**mrF** or **mR**)

For binary code (Numeric)

For permutation code (TSP)

- The **setVariables** method, after setting **popSize**, sets the variables (**pC**, **pM**) to either the values of (**uXp**, **mrF**) or those of (**XR**, **mR**) depending on whether the problem to solve is numerical optimization or TSP, respectively
  - While (**pC**, **pM**) are the parameters of the search algorithm, the user only gives the values of (**uXp**, **mrF**) or (**XR**, **mR**)

For Numeric For TSP

resolution : number of bits for binary-encoding

Adding Genetic Algorithm

- **`displaySetting`** shows the population size and the parameter values used by the genetic operators
  - **`resolution`**, **`uXp`**, and **`mrF`** for a numerical optimization problem
  - **`XR`** and **`mR`** for a TSP

- The genetic algorithm itself is implemented as the **`run`** method of the class
  - (note) Among the many different genetic algorithms, our implementation is just one of them. In other words, there are diverse ways to implement GA
  - The population is a list of individuals
  - An individual is a list of a pair: [*fitness*, *chromosome*]
    - That is, the first item in the list storing each solution is the **fitness** score.
  - The **`run`** method makes calls to some methods of **`Problem`** class and other methods within the **`GA`** class (**`run`** use both **`Problem`** class and **`GA`** class methods)

Adding Genetic Algorithm

- **`run(self, p)`**:
  - Generates an initial population randomly (**`p.initializePop`**)
  - Evaluates individuals in the initial population and identifies the best one (**`self.evalAndFindBest`**)
  - Until termination, keeps applying the genetic operators (**`self.select-Parents`**, **`p.crossover`**, **`p.mutation`**), updating the population (= generate next solution set), and updating the best-so-far individual (**`self.eva-lAndFindBest`**)
  - Converts the best individual found to a form containing only the solution part (**`p.indToSol`**)
    - Since each individual = [fitness, solution], the **`indToSol`** method is implemented to extract only the solution part.
    - **`ind = individual`**
  - Stores the best solution (**`p.storeResult`**)

Adding Genetic Algorithm

- **`evalAndFindBest(self, pop, p)`**:
  - Evaluates each individual in the population **`pop`** (**`p.evalInd`**), identifies the best one, and returns it
- **`selectParents(self, pop)`**:
  - Performs binary tournament selection twice (**`self.selectTwo`**, **`self.binaryTournament`**) and returns the selected parents
  - Selects two solutions by performing the process of randomly selecting two parents using **`selectTwo`** and then choosing the parent with the lower fitness score via **`binaryTournament`**, repeating this process twice.
- **`selectTwo(self, pop)`**:
  - Selects two random individuals in **`pop`** and returns them
- **`binaryTournament(self, ind1, ind2)`**:
  - Returns the better solution between the two solutions (**`ind1`**, **`ind2`**).
  - Returns the individual (solution) whose fitness score (which is **`ind1[0]`**, **`ind2[0]`**) is "smaller".
    - The **fitness score** is the result of the evaluation, and since we are solving a **minimization** problem, the solution with the "smaller" fitness score is the "better" solution.

Changes to 'Setup' Class

- A variable named **`resolution`** is newly added
  - It represents the length of the binary string for each variable for a numeric optimization problem
  - It is referenced by the methods in both the classes **`Optimizer`** and **`Problem`**

- The **`setVariables`** method is accordingly changed

- Changes to the main program is minimal
  - **`readPlan`** is revised to read in more parameter values such as the population size, crossover rate, mutation rate, etc.
  - **`createOptimizer`** is revised to include GA as its 6th optimizer

```
optimizers = { 1: 'SteepestAscent()',
               2: 'FirstChoice()',
               3: 'Stochastic()',
               4: 'GradientDescent()',
               5: 'SimulatedAnnealing()',
               6: 'GA()' }
```

Changes to 'Problem' Class

- Many methods are added to the classes **`Numeric`** and **`Tsp`** to support operations needed to conduct the search by genetic algorithm

- We first describe the five methods that are commonly added to both **`Numeric`** and **`Tsp`**
  - **`initializePop(self, size)`**:
    - Makes a population of given **`size`** with randomly generated individuals, and returns it (= 랜덤하게 size 수 만큼의 솔루션을 생성)
    - In **`Numeric`**, the individual chromosome for GA is represented by a binary string that is different from that used by other algorithms (**`self.randBinStr`**)
    - In **`Tsp`**, the individual chromosome for GA is represented by permutation code that is also used by other search algorithms (**`self.randomInit`**)
      - ✓ permutation code = uses the result of permuting city numbers, represented as **`int`**, as the solution.

Changes to 'Problem' Class

- **`evalInd(self, ind)`**:
  - Evaluates the chromosome of **`ind`** and records the fitness value (**`self.evaluate`**)
  - In **`Numeric`**, however, the binary string must be decoded before it can be evaluated (**`self.decode`**)
  - The calculated fitness score is stored at index 0 of the passed-in **individual**.

- **`crossover(self, ind1, ind2, pC)`**:
  - Performs crossover to parents and returns the resulting children
  - In **`Numeric`**, a uniform crossover is performed interpreting **`pC`** as the swap probatility **`uXp`** (**`self.uXover`**)
    - ✓ Decides whether or not to perform crossover at each position.
  - In **`Tsp`**, an ordered crossover is performed interpreting **`pC`** as the crossover rate **`XR`** (**`self.oXover`**)
    - ✓ Decides whether or not to perform the crossover operation itself.
    - ✓ If performed, a random range is generated, and **ordered crossover** is carried out based on this range.

Changes to 'Problem' Class

- **`mutation(self, ind, pM)`**:
  - Performs mutation to **`ind`** and returns it (Performs solution variation on a single solution)
  - In **`Numeric`**, a bit-flip mutation is performed interpreting **`pM`** as the factor **`mrF`** to adjust the mutation rate
    - ✓ Randomly performs bit flip for each index of the solution.
  - In **`Tsp`**, an inversion operation is performed interpreting **`pM`** as the mutation rate **`mR`** (**`self.inversion`**)
    - ✓ Generates a random start and end position and Inverts the values within that range (mutate by inversion).

- **`indToSol(self, ind)`**:
  - Converts an individual to a form containing only the solution part
    - ✓ Since the input argument **`ind`** stores [fitness, solution], it returns everything except the fitness.
    - ✓ In other words, `return self.decode(ind[1])`
    - ✓ Note that **`indToSol`** function called in **`GA.run`** use `bestSolution = p.indToSol(best)`
  - In **`Numeric`**, the chromosome is decoded and then returned (**`self.decode`**)
  - In **`Tsp`**, just the chromosome part of **`ind`** is returned

Changes to 'Problem' Class

- We now describe the methods that are added only to **`Numeric`**
  - **`randBinStr(self)`**:
    - Generates a random binary string of a predetermined length (**`self._resolution`**), and returns it
  - **`decode(self, chromosome)`**:
    - Decodes each variable in **`chromosome`** to its decimal value (**`self.binary-ToDecimal`**), concatenates them to a solution form, and returns it
  - **`binaryToDecimal(self, binCode, l, u)`**:
    - Decodes **`binCode`** to a decimal value taking the domain and resolution into account, and returns it
  - **`uXover(self, chrInd1, chrInd2, uXp)`**:
    - Performs uniform crossover to two chromosomes **`chrInd1`** and **`chrInd2`**, and returns the resulting chromosomes

Changes to 'Problem' Class

- We now describe one method that is added only to **`Tsp`**
  - **`oXover(self, chrInd1, chrInd2)`**:
    - Performs ordered crossover to two chromosomes **`chrInd1`** and **`chrInd2`**, and returns the resulting chromosomes

Code sample for GA

```
def run(self, p):
    # Population is a list of individuals
    # individual: [fitness, chromosome]
    pop = p.initializePop(self._popSize)
    best = self.evalAndFindBest(pop, p)
    numEval = p.getNumEval()
    whenBestFound = numEval
    while numEval < self._limitEval:
        newPop = []
        i = 0
        while i < self._popSize:
            par1, par2 = self.selectParents(pop)
            ch1, ch2 = p.crossover(par1, par2, self._pC)
            newPop.extend([ch1, ch2])
            i += 2
        newPop = [p.mutation(ind, self._pM) for ind in newPop]
        pop = newPop
        newBest = self.evalAndFindBest(pop, p)
        numEval = p.getNumEval()
        if newBest[0] < best[0]:
            best = newBest
            whenBestFound = numEval
    self._whenBestFound = whenBestFound
    bestSolution = p.indToSol(best)
    p.storeResult(bestSolution, best[0])
```

Generates the initial population set.

Selects two soln. from current pop.

Crossover on two parents to generate two new child soln.

Applies the mutation operation to the generated solutions (ind=individual solution)

Updates the overall best solution found so far.

Code sample for Numeric problems

```
def initializePop(self, size): # Make a population of given size
    pop = []
    for i in range(size):
        chromosome = self.randBinStr()
        pop.append([0, chromosome])
    return pop

def randBinStr(self):
    k = len(self._domain[0]) * self._resolution
    chromosome = []
    for i in range(k):
        allele = random.randint(0, 1)
        chromosome.append(allele)
    return chromosome
```

Generate initial solution set (Each solution is encoded as a 0/1 binary string)

Fitness score

k = {number of variables} * {length of the binary encoding string to represent each variable}

Randomly generate an initial solution (Each solution is encoded as a 0/1 binary string)

The r-bit binary strings representing N variables are all concatenated and stored as a single list

Code sample for Numeric problems

```
def decode(self, chromosome):
    r = self._resolution
    low = self._domain[1]  # list of lower bounds
    up = self._domain[2]   # list of upper bounds
    genotype = chromosome[:]
    phenotype = []
    start = 0
    end = r   # The following loop repeats for # variables
    for var in range(len(self._domain[0])):
        value = self.binaryToDecimal(genotype[start:end],
                                     low[var], up[var])
        phenotype.append(value)
        start += r
        end += r
    return phenotype

def binaryToDecimal(self, binCode, l, u):
    r = len(binCode)
    decimalValue = 0
    for i in range(r):
        decimalValue += binCode[i] * (2 ** (r - 1 - i))
    return l + (u - l) * decimalValue / 2 ** r
```

Function to convert each 0/1 binary list (chromosome) into a list of real (floating-point) values

The r-bit binary strings representing N variables are all concatenated and stored as a single 1D list. To extract the binary string corresponding to each variable, you must **index using [start:end]** at the relevant positions

Convert the value encoded as a 0/1 binary string into a real (floating-point) value (A real value is needed when evaluating the solution/fitness)

Code sample for Numeric problems

```
def crossover(self, ind1, ind2, uXp):
    # pC is interpreted as uXp# (probability of swap)
    chr1, chr2 = self.uXover(ind1[1], ind2[1], uXp)
    return [0, chr1], [0, chr2]

def uXover(self, chrInd1, chrInd2, uXp): # uniform crossover
    chr1 = chrInd1[:]  # Make copies
    chr2 = chrInd2[:]
    for i in range(len(chr1)):
        if random.uniform(0, 1) < uXp:
            chr1[i], chr2[i] = chr2[i], chr1[i]
    return chr1, chr2

def mutation(self, ind, mrF):  # bit-flip mutation
    # pM is interpreted as mrF (factor to adjust mutation rate)
    child = ind[:]    # Make copy
    n = len(ind[1])
    for i in range(n):
        if random.uniform(0, 1) < mrF * (1 / n):
            child[1][i] = 1 - child[1][i]
    return child
```

Operation that combines two solutions through crossover

For each index of the binary string representing a solution, perform the crossover operation according to a probability

For each index of the binary string representing a single solution, perform the bit-flip operation according to a probability

Code sample for TSP problems

```python
def initializePop(self, size): # Make a population of given size
    n = self._numCities        # n: number of cities
    pop = []
    for i in range(size):
        chromosome = self.randomInit()
        pop.append([0, chromosome])
    return pop
```

Initially, generate 'size' number of initial random solutions

```python
def randomInit(self):   # Return a random initial tour
    n = self._numCities
    init = list(range(n))
    random.shuffle(init)
    return init
```

```python
def mutation(self, ind, mR): # mutation by inversion
    # pM is interpreted as mR (mutation rate for inversion)
    child = ind[:]  # Make copy
    if random.uniform(0, 1) <= mR:
        i, j = sorted([random.randrange(self._numCities)
                       for _ in range(2)])
        child[1] = self.inversion(child[1], i, j)
    return child
```

Reverse the order of values within the range

Code sample for TSP problems

- Decide whether or not to perform crossover based on the **`XR`** (Crossover Rate) probability
  - If not performed, return the two solutions (**`ind1, ind2`**) as is (unchanged)

```
def crossover(self, ind1, ind2, XR):
    # pC is interpreted as XR (crossover rate)
    if random.uniform(0, 1) <= XR:
        chr1, chr2 = self.oXover(ind1[1], ind2[1])
    else:
        chr1, chr2 = ind1[1][:], ind2[1][:]  # No change
    return [0, chr1], [0, chr2]
```

Code sample for TSP problems

```python
def oXover(self, chrInd1, chrInd2):  # Ordered Crossover
    chr1 = chrInd1[:]
    chr2 = chrInd2[:]  # Make copies
    size = len(chr1)
    a, b = sorted([random.randrange(size) for _ in range(2)])
    holes1, holes2 = [True] * size, [True] * size
    for i in range(size):
        if i < a or i > b:
            holes1[chr2[i]] = False
            holes2[chr1[i]] = False
    # We must keep the original values somewhere
    # before scrambling everything
    temp1, temp2 = chr1, chr2
    k1, k2 = b + 1, b + 1
    for i in range(size):
        if not holes1[temp1[(i + b + 1) % size]]:
            chr1[k1 % size] = temp1[(i + b + 1) % size]
            k1 += 1
        if not holes2[temp2[(i + b + 1) % size]]:
            chr2[k2 % size] = temp2[(i + b + 1) % size]
            k2 += 1
    # Swap the content between a and b (included)
    for i in range(a, b + 1):
        chr1[i], chr2[i] = chr2[i], chr1[i]
    return chr1, chr2
```

deep copy

Randomly generate range a, b

Mark numbers not included in the selected range as **False**

Starting from index **b+1**, copy the numbers not included in the selected range in **sequential order**

- chr1 :
  - Values within the [a,b] range are taken from chr2
  - Values outside the range are taken from chr1
- chr2 :
  - Values within the [a,b] range are taken from chr1
  - Values outside the range are taken from chr2

Swap the numbers within the set range

Experiments

- Solve the given 6 problems using all algorithms implemented so far, and compare their performance.

Experiments

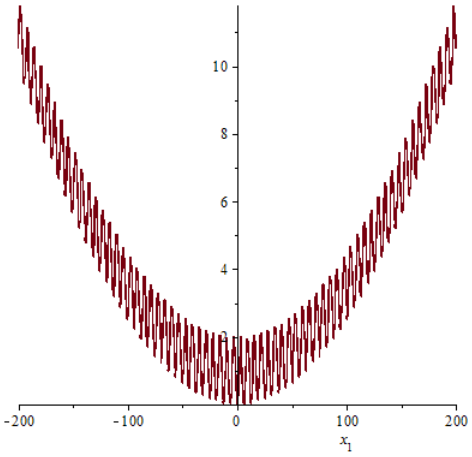
- We solve a few numerical optimization problems and TSPs using various search algorithms made available in our optimization tool and compare their performances
- We try three numerical optimization problems all of which are five dimensional with the search space of $-30 \le x_i \le 30$ for $1 \le i \le 5$
  - Convex function:

$$(x_1 - 2)^2 + 5(x_2 - 5)^2 + 8(x_3 + 8)^2 + 3(x_4 + 1)^2 + 6(x_5 - 7)^2$$

  - Griewank function:

$$1 + \frac{1}{4000}\sum_{i=1}^{5} x_i^2 - \prod_{i=1}^{5} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

    with a global minimum of 0 at all zeros



One-dimensional Griewank function

Experiments

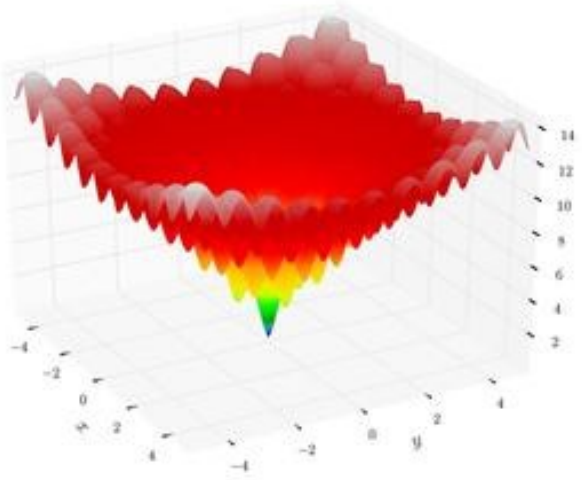
– Ackley function:

$$20 + e - 20\exp\left(-\frac{1}{5}\sqrt{\frac{1}{5}\sum_{i=1}^{5} x_i^2}\right) - \exp\left(\frac{1}{5}\sum_{i=1}^{5}\cos(2\pi x_i)\right)$$

with a global minimum of 0 at all zeros

- We also try three versions of TSPs with different numbers of cities located in a 100×100 square
  – We use TSP-N as the name of a TSP with N cities, where N is 30, 50, and 100

Two-dimensional Ackley function



Experimental Setting

- Step size of mutation for the three hill climbers steepest-ascent, first-choice, and stochastic: 0.01
- For the gradient descent algorithm:
  - Update rate: 0.01
  - Size of dx for calculating gradient: 10^{-4}

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$$

$$\frac{df(x)}{dx} = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx}$$

- For the two hill climbers first-choice and stochastic:
  - Consecutive iterations allowed for no improvement: $1{,}000$

- For the two metaheuristic algorithms:
  - The total number of evaluations until termination: $500{,}000$

Experimental Setting

- Population size of GA: 100

- GA for numerical optimization:
  - The length of binary chromosome per variable: 10
  - Swap probability for uniform crossover: 0.2
  - Multiplication factor to $1/l$ for mutation (l: length of chromosome): 1
    - Thereby, the probability to **`mutate`** for each index is $=1*1/l$

- GA for TSP:
  - Crossover (ordered crossover) rate: 0.5
  - Mutation (inversion) rate: 0.2

Experimental Results

- The numbers shown in the table are the averages of 10 experiments
  - All the hill climbers are randomly restarted by 10 times in each experiment
- The four numbers in each cell represent the following:
  - Average objective value (left top)
  - Best objective value found (left bottom)
  - Average number of evaluations (right top)
  - Average iteration of finding the best solution (right bottom)
    (only for GA and Simulated Annealing)
- Bold-faced letters indicate the best result among different optimizers
- For TSPs, the results are compared with those obtained by the nearest-neighbor algorithm, which starts from a random city and keeps visiting the one that is the closest (무작위로 선택된 도시에서 출발하고 , 가장 가까운 도시를 방문하는 전략)

Experimental Results

- Convex function:
  - All the algorithms except GA found the optimal solution
  - Gradient descent reaches the optimum the fastest
    (SA is faster but it does not terminate automatically)
  - First-choice is faster than steepest-ascent

- Griewank function:
  - GA performs much better than the others
  - SA performs worse than the hill climbers

- Ackley function:
  - GA is much better than the others
  - SA performs worse than the hill climbers

Experimental Results

- TSPs:
  - Steepest-ascent is worse than nearest-neighbor
  - Stochastic shows the best average performance with TSP-50
    - But it took too many iterations to solve TSP-100
  - SA shows the overall best performance
  - GA does not show any competitive performance when the problem size is small (TSP-30)

- The quality of the solution found by metaheuristic algorithms is better than that by hill climbers for most problems
- A hill climber should be the choice if the problem is convex
- The results reported here are obtained without enough parameter tuning (more careful investigation is needed)

Results of Numerical Optimization

	Convex		Griewank		Ackley	
Steepest Ascent	0.0 0.0	774,692	0.260 0.108	67,144	17.832 14.029	12,182
First Choice	0.0 0.0	274,521	0.254 0.064	38,824	18.214 14.108	14,377
Stochastic	0.0 0.0	2,088,171	0.218 0.096	387,008	18.879 16.729	141,156
Gradient Descent	**0.0** **0.0**	199,935	0.216 0.118	856,635	17.447 8.101	5,234
Simulated Annealing	0.0 0.0	500,000 63,565	0.367 0.145	500,000 18,252	19.319 18.940	500,000 5,541
GA	3.920 0.766	500,000 227,140	**0.036** **0.015**	500,000 220,390	**0.214** **0.141**	500,000 173,900

- The four numbers in each cell represent the following:
  - Average objective value (left top)
  - Best objective value found (left bottom)
  - Average number of evaluations (right top)
  - Average iteration of finding the best solution (right bottom) (only for GA and Simulated Annealing)

Results of Combinatorial Optimization (TSP)

	TSP-30		TSP-50		TSP-100	
Steepest Ascent	593 525	6,846	782 678	20,211	1,223 1,145	88,254
First Choice	424 408	30,154	578 561	57,207	903 869	131,450
Stochastic	422 408	670,952	**570** 561	2,395,138	872 840	10,903,041
Nearest Neighbor	509 455		694 646		958 918	
Simulated Annealing	**412** **408**	500,000 28,831	577 559	500,000 43,962	**829** **804**	500,000 69,084
GA	658 635	500,000 245,940	584 **558**	500,000 226,840	854 822	422,650

- The four numbers in each cell represent the following:
  - Average objective value (left top)
  - Best objective value found (left bottom)
  - Average number of evaluations (right top)
  - Average iteration of finding the best solution (right bottom) (only for GA and Simulated Annealing)