Search Algorithms: Object-Oriented Implementation (Part E)

Contents

- Conventional vs. AI Algorithms
- Local Search Algorithms
- Genetic Algorithm
- Implementing Hill-Climbing Algorithms
- Defining 'Problem' Class
- Adding Gradient Descent
- Defining 'HillClimbing' Class
- Adding More Algorithms and Classes
- Adding Genetic Algorithm
- Experiments

Adding More Algorithms and Classes

- (TODO) Add 3 new Search Algorithms to the existing code:
  - random-restart
  - stochastic hill climbing
  - simulated annealing

  } *Code implementation will be covered*

- (TODO) Also, modify the class structure for expandability/extensibility
  - To easily add diverse types of algorithm

Review (1/2)

- From 09 Search Algorithm (A) previously...
  - Search algorithms derived from Hill Climbing (= steepest ascent) :

New algorithm to be implemented

- Stochastic hill climbing:
  - ✓ Chooses at random from among the uphill moves with probability proportional to steepness
  - ✓ Assigns probability to each candidate state/solution proportional to its steepness (gradient), and selects one based on this probability.
  - ✓ The rest of the process is the same as First-Choice.
- Random-restart hill climbing:
  - ✓ Conducts a series of hill-climbing searches from randomly generated initial states
  - ✓ Repeats hill-climbing multiple times by varying the *starting point* (initial state), and selects the best solution among them

Already implemented

- First-choice (simple) hill climbing:
  - ✓ Generates successors randomly until one is found that is better than the current state
  - ✓ Randomly selects a candidate state and selects it if it is better than the `current state` (simple calculation).

Review (2/2)

- From 09 Search Algorithm (A) previously...
  - Simulated Annealing Search Algorithm:

New algorithm to be implemented

- Selects a *bad solution* with a **random probability** and, through this, can find a **better solution**.
- **Note:** *Gradually reduce* the probability of selecting a bad solution.

[Temperature change schedule] Passed as input, a schedule that changes the temperature from high to low (close to 0). It's stored as a list of what the temperature will be at each time-step.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE – current.VALUE
    if ΔE < 0 then current ← next
    else current ← next only with probability e^(−ΔE/T)
```

Calculate the current state from the initial state

Randomly select the next state

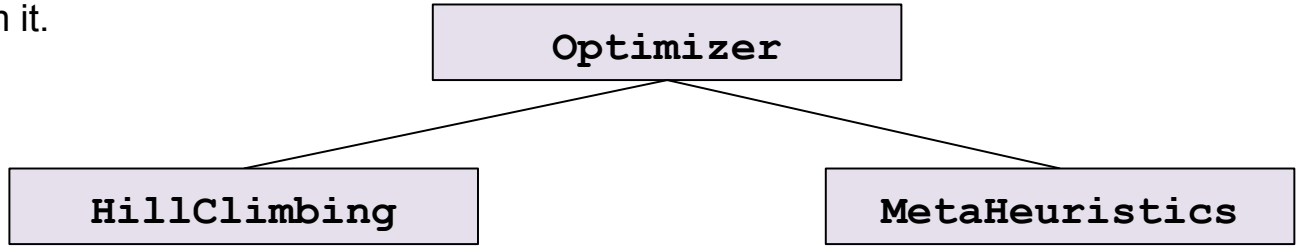
Calculate the difference in objective function between the two states (= energy difference)

Assumes a minimization problem ➔ Lower value is better

Selects a bad state probabilistically. Here, the probability varies based on the difference between the two solutions and the temperature (T)

Adding More Algorithms and Classes

- (TODO) Add 3 new Search Algorithms to the existing code: stochastic hill climbing, random-restart, simulated annealing
  - *stochastic HC* and *random-restart* are algorithms derived from **Hill Climbing**..
    In contrast, *simulated annealing* is a type of Meta-Heuristics algorithm.
    - Summary of solution algorithm types:
      - ✓ HillClimbing derived search algo. : 1) steepest ascent, 2) first-choice, 3) stochastic, 4) random-restart, 5) gradient descent
      - ✓ MetaHeuristics derived search algo. : 1) simulated annealing, 2) GA (*next topic*)
  - **A more general class hierarchy is needed** to encompass diverse types of search algorithms.
  - **Class Structure Refactoring:** To do this, define a *new class* called Optimizer and modify the code so that the *existing* HillClimbing class and the *newly defined* MetaHeuristics class inherit from it.

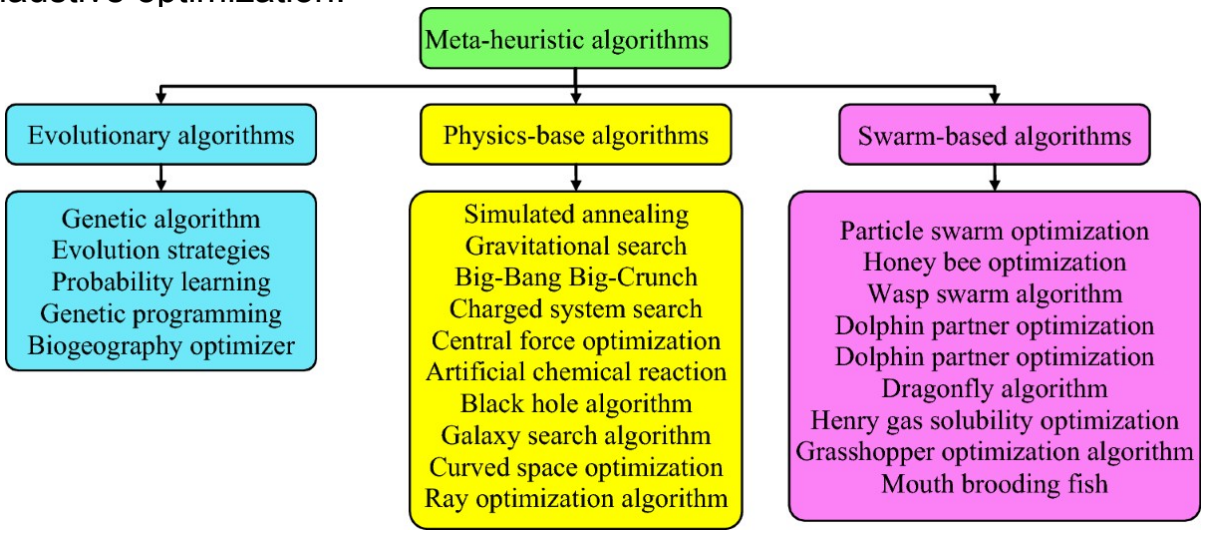


Adding More Algorithms and Classes

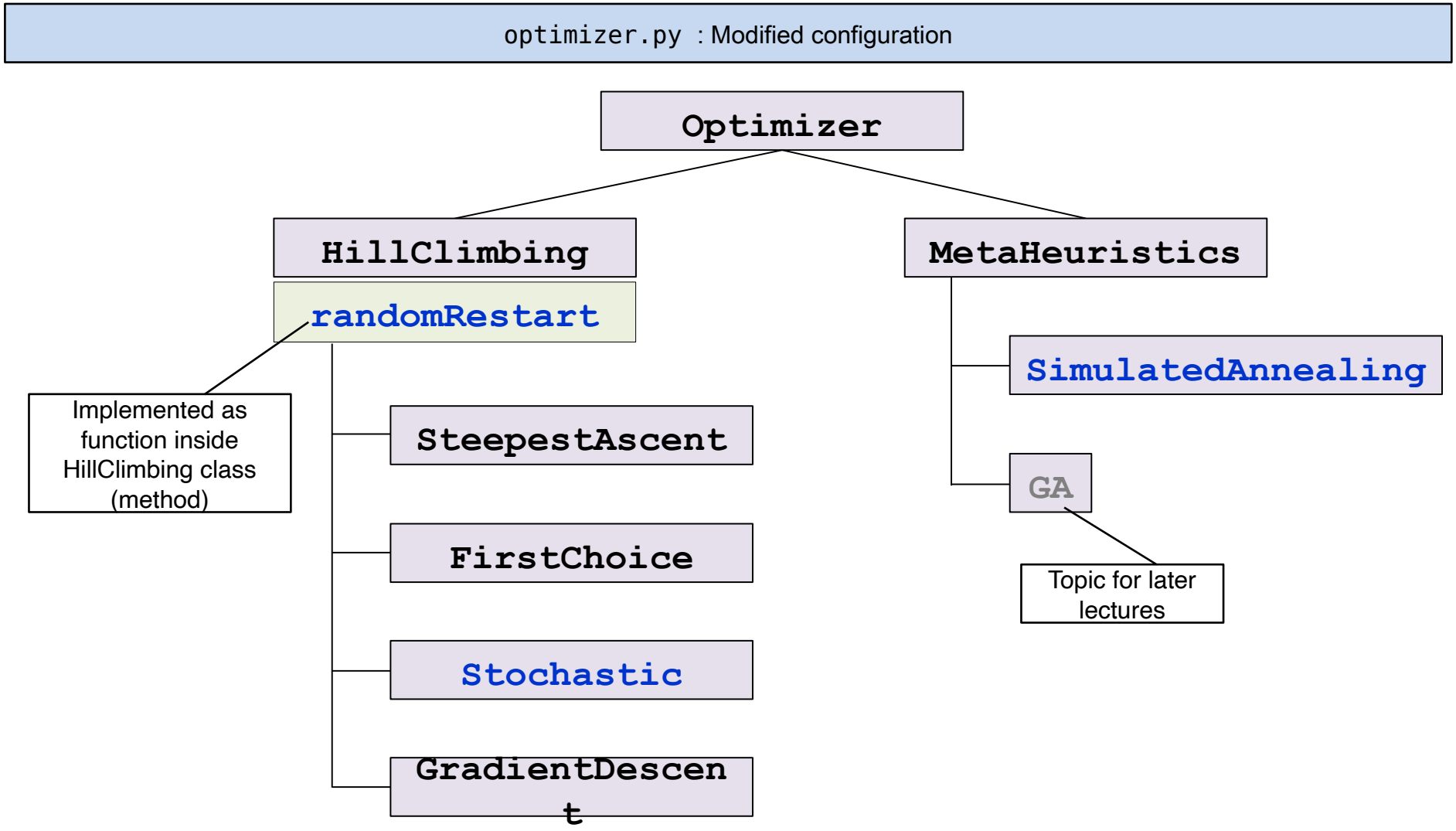
- (TODO) Add 3 new Search Algorithms to the existing code: stochastic hill climbing, random-restart, simulated annealing
  - Since there are 5 algorithms of the `HillClimbing` type, 5 sub-classes can be created, and each can inherit the `HillClimbing` class to be implemented.
  - However, we can implement only 4 sub-classes and design them so that the random-restart technique is executed by all classes that inherit `HillClimbing` through an implementation within the `HillClimbing` class.
    - The techniques: **`steepest-ascent`**, **`first-choice`**, **`stochastic`**, **`gradient descent`** are all algorithms that define how to find the **next solution** and which one to select among them.
    - **`random-restart`** is a technique that does not define how to find the **next solution**, but rather defines repeatedly performing hill climbing using a random initial solution and selecting the best solution among them. Therefore, it can be applied to all 4 remaining search algorithms.

Adding More Algorithms and Classes

- (TODO) Add 3 new Search Algorithms to the existing code: stochastic hill climbing, random-restart, simulated annealing
  - **`MetaHeuristics`** will have two sub-classes: **`SimulatedAnnealing`** and **`GA`** (Note: GA is a technique we will learn about later).
    - Note: Meta-Heuristics refers to an **"algorithm that can be applied universally to any problem,"** not just limited to a specific problem.
    - Note: A **Heuristic** algorithm refers to an algorithm designed to solve a given problem **more quickly and efficiently**, but its accuracy may be lower compared to algorithms based on exhaustive optimization.



Adding More Algorithms and Classes



Adding More Algorithms and Classes

- All **Hill-Climbing** algorithms are fundamentally executed through **randomRestart**, and they operate by repeating the execution of the selected algorithm for a specified number of times (e.g., repeating the execution a number of times influences the initial solution generated randomly) and selecting the best solution among them.
  - Each hill climbing algorithm itself has its own **run** method, but if the **`randomRestart`** method is executed, inherited from the `HillClimbing` class, the `randomRestart` method operates by executing the **run** method defined in the algorithm selected by the user.
  - Therefore, the **`randomRestart`** method, which is commonly used by a number of solution algorithms, is implemented in the **`HillClimbing`** class (which is the superclass of all hill-climbing algorithms) and is inherited by the sub-classes.

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

⋮

```
if 1 <= aType <= 4:
    alg.randomRestart(p)
else:
    alg.run(p)
```

HillClimbing Class

```
def randomRestart(self, p):
    i = 1
    self.run(p)
    bestSolution = p.getSolution()
    bestMinimum = p.getValue()    #
    numEval = p.getNumEval()
    while i < self._numRestart:
        self.run(p)
```

```
class SteepestAscent(HillClimbing):
    def displaySetting(self):
        print()
        print("Search Algorithm: Steepest
        print()
        HillClimbing.displaySetting(self)

    def run(self, p):
```

In alg=SteepestAscent(), run method under SteepestAscent class is extecuted

Adding More Algorithms and Classes

- When a **hill-climbing algorithm** is arbitrarily selected and executed:
  - A **class instance** corresponding to the algorithm selected by the user is created and stored in **`alg`**.
  - next, **`alg.randomRestart`** is executed, calling `randomRestart`.
  - The `randomRestart` method works by repeatedly executing the **`self.run`** method of the selected hill-climbing algorithm (using a different random initial value each time).

- Conversely, in the case of **metaheuristic** objects, they operate by creating an object of the selected class and immediately executing the **`run`** method.

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

Defined in the form of a Dictionary Value, which can be passed as an argument to the Python eval function to create a class instance (by calling the class constructor).

```
def randomRestart(self, p):
    i = 1
    self.run(p)
    bestSolution = p.getSolution()
    bestMinimum = p.getValue()    #
    numEval = p.getNumEval()
    while i < self._numRestart:
        self.run(p)
```

```
if 1 <= aType <= 4:
    alg.randomRestart(p)
else:
    alg.run(p)
```

Adding More Algorithms and Classes

- **Modifying the User Interface** as the number of algorithm types and associated parameters increases.
  - **Existing User Interface,** the user inputs the following **3 pieces of information** in the terminal:
    - Problem type: `Numeric / TSP`
    - Solution Algorithm: `SA / FC / GradDesc`
    - Filename where the problem is defined: `problem/xxx.txt`
  - **Modified User Interface,** the user inputs the following **1 piece of information** in the terminal:
    - **The name of the text file** where all parameters related to algorithm execution (= setup information) are

```
def readPlan():
    fileName = input("Enter the file name of experimental setting: ")
    infile = open(fileName, 'r')
```

    - Contents to be included in the text file:
      - ✓ Problem Type
      - ✓ Solution Algorithm
      - ✓ Filename where the problem is defined
      - ✓ Parameter Values: `delta, limitStuck, dx, numRestart, limitEval, numExp`
      - ✓ (File example: next slide...)

Adding More Algorithms and Classes

- **Modifying the User Interface** as the number of algorithm types and associated parameters increases.
  - **Modified User Interface,** the user inputs the following **1 piece of information** in the terminal
    - **The name of the text file** where all setup information are defined.
    - Example text files ("#" line is a comment and will not be read):

```
# Select the problem type:
#    1. Numerical Optimization
#    2. TSP
  Enter the number (pType) : 2
#
#   Enter the name of the file : problem/Convex.txt
#   Enter the name of the file : problem/Griewank.txt
#   Enter the name of the file : problem/Ackley.txt
#   Enter the name of the file : problem/tsp30.txt
#   Enter the name of the file : problem/tsp50.txt
  Enter the name of the file : problem/tsp100.txt
#
# Select the search algorithm:
#  Hill Climbing algorithms:
#    1. Steepest-Ascent
#    2. First-Choice
#    3. Stochastic
#    4. Gradient Descent
#  Metaheuristic algorithms:
#    5. Simulated Annealing
  Enter the number (aType) : 2
#
```

```
# If you are solving a function optimization problem,
#   what should be the step size for axis-parallel mutation?
  Mutation step size (delta) : 0.01
#
# If your algorithm choice is 2 or 3,
#   what should be the number of consecutive iterations without improvement?
  Give the number of iterations (limitStuck) : 1000
#
# If your algorithm choice is 4 (gradient descent),
# what should be the update step size and increment for derivative?
  Update rate for gradient descent (alpha) : 0.01
  Increment for calculating derivative (dx) : 10 ** (-4)
#
# If you want a random-restart hill climbing,
#   enter the number of restart.
# Enter 1 if you do not want a random-restart.
  Number of restarts (numRestart) : 10
#
# If you are running a metaheuristic algorithm,
#   what should be the total number of evaluations until temination?
  Enter the number (limitEval) : 100000
#
# Enter the total number of experiments
   Enter the number (numExp) : 10
```

Adding More Algorithms and Classes

- The following **4 variables** were added to the *Optimizer* class or its sub-classes:
  - **`(1) numExp:`**
    - total number of experiments to be conducted.
    - added to the **`Optimizer`** since it is commonly applied to all solution algorithms.
    - In hill-climbing techniques it determines `randomRestart` method execution, in meta-heuristics it determines `run` method execution

Added Details

- ✓ Hill-Climbing : Decides how many times the *randomRestart* method is called. The number of **random restarts** performed after *randomRestart* is called is determined by *self.numRestart*.
- ✓ Meta-Heuristics : Decides how many times the *run* method is called.

    - In the `main.py` code...

```
numExp = alg.getNumExp()
for i in range(1, numExp):
    if 1 <= aType <= 4:
        alg.randomRestart(p)
    else:
        alg.run(p)
```

- Because the **first experiment (i=0)** is handled separately, the **`for`** loop executes only ***`numExp`*** **`- 1`** times
- The **randomRestart** function executes the algorithm repeatedly for **`self.numRestart`** times. Therefore, the **Hill Climbing** technique is executed a total of **`numExp * self.numRestart`** times.

Adding More Algorithms and Classes

- The following **4 variables** were added to the *Optimizer* class or its sub-classes:
  - **`(2) limitEval`**:
    - maximum number of evaluations until termination for metaheuristic algorithms (Added to the **`MetaHeuristics`** class)

    - `run` method in `simulatedAnnealing` class...

```
def run(self, p):
    :
    while True:
        t = self.tSchedule(t)   # Follow annealing schedule
        if t == 0 or i == self._limitEval:
            break
    :
```

The temperature (`t`) is set to decrease gradually, but since it takes too long to reach 0 or may not reach 0 if set too aggressively, this variable (`limitEval`) is added to **limit the number of algorithm executions** when the temperature is very close to 0.

```
def tSchedule(self, t):
    # 즉, t' = t * 0.999
    return t * (1 - (1 / 10**4))
```

Temperature scheduling setup

Adding More Algorithms and Classes

- The following **4 variables** were added to the *Optimizer* class or its sub-classes:
  - **`(3) whenBestFound`**:
    - the number of iterations taken until the best solution has first been found
      (Added to the **`MetaHeuristics`** class)
    - It stores how many total **iterations** were executed until the *best solution* was found.

    - `run` method in `simulatedAnnealing` class...

```
def run(self, p):
        ⋮
    while True:
        t = self.tSchedule(t)   # Follow annealing schedule
        if t == 0 or i == self._limitEval:
            break
        ⋮
            whenBestFound = i   # Record when best was found
    self._whenBestFound = whenBestFound
```

Adding More Algorithms and Classes

- The following **4 variables** were added to the *Optimizer* class or its sub-classes:
  - **`(4) numRestart`**:
    - number of random restarts for the random-restart algorithm (Added to **`HillClimbing` class**)
    - This variable determines **how many times** the random restart will be repeated when *randomRestart* is called.

    - `randomRestart` method in `HillClimbing` class...

```
while i < self._numRestart:
    self.run(p)
```

Calls the **run** method of the class for the solution algorithm selected by the user.

Adding More Algorithms and Classes

- The **`aType`** member variable is added to the **`Setup`** class.
  - Since this variable is used by both the superclass **`Problem`** defined in *problem.py* and the superclass **`Optimizer`** defined in *optimizer.py*, the **`aType`** variable is added to the **`Setup`** class, which is inherited by both.
    - It is referenced by the revised **`report`** method of **`Problem`**
    - It is also referenced by **`displaySetting`** of **`Optimizer`**

main.py

```
# Select the search algorithm:
#  Hill Climbing algorithms:
#    1. Steepest-Ascent
#    2. First-Choice
#    3. Stochastic
#    4. Gradient Descent
#  Metaheuristic algorithms:
#    5. Simulated Annealing
   Enter the number (aType) : 5
```

```
for i in range(len(parNames)):
    line = lineAfterComments(infile)
    if parNames[i] == 'pFileName':
        # 문제의 특성이 저장된 파일의 이름은 텍스트로 취급/처리
        parameters[parNames[i]] = line.rstrip().split(':')[-1][1:]
    else:
        # 나머지 정보는 숫자로 취급/처리
        parameters[parNames[i]] = eval(line.rstrip().split(':')[-1][1:])
```

```
alg.setVariables(parameters)
```

```
def setVariables(self, parameters): # HillClimbing.setVariables
    Optimizer.setVariables(self, parameters)
    self._l
    self._n
```

```
def setVariables(self, parameters): # Optimizer.setVariables
    Setup.setVariables(self, parameters)
    self._p
    self._r
```

```
def setVariables(self, parameters): # Setup.setVariable
    self._aType = parameters['aType']
    self._delta = parameters['delta']
    self._alpha = parameters['alpha']
    self._dx = parameters['dx']
```

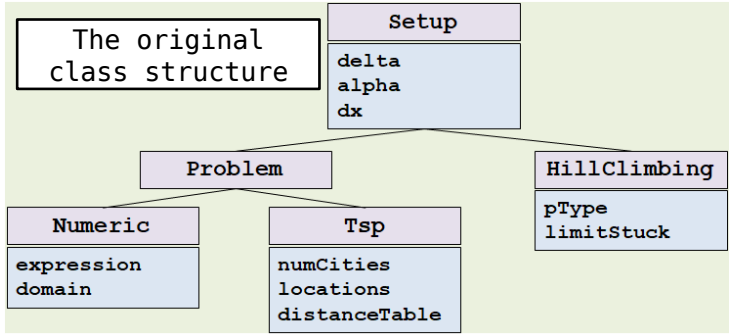
Adding More Algorithms and Classes

- The code was modified to reference the **`pType`** variable in the **`displaySetting`** method of the **`Optimizer`** class. Due to this change, the **`pType`** variable was **moved** from the **`HillClimbing`** class to the **`Optimizer`** class.

```
class Optimizer(Setup):
    def __init__(self):
        Setup.__init__(self)
        self._pType = 0   # Type of problem
        self._numExp = 0  # Total number of experiments

    def setVariables(self, parameters): # Optimizer.setVariables
        Setup.setVariables(self, parameters)
        self._pType = parameters['pType']
        self._numExp = parameters['numExp']

    def getNumExp(self):
        return self._numExp

    def displayNumExp(self):
        print()
        print("Number of experiments:", self._numExp)

    def displaySetting(self): # Optimizer.displaySetting
        if self._pType == 1 and self._aType != 4 and self._aType != 6:
            print("Mutation step size:", self._delta)
```

Adding More Algorithms and Classes

the location of the aType variable and the newly added 4 variables (numExp, numRestart, limitEval, whenBestFound) within the class hierarchy.



Setup
aType
delta
alpha
dx

Problem
solution
value
numEval

Optimizer
pType
numExp

Numeric
expression
domain

Tsp
numCities
locations
distanceTable

HillClimbing
limitStuck
numRestart

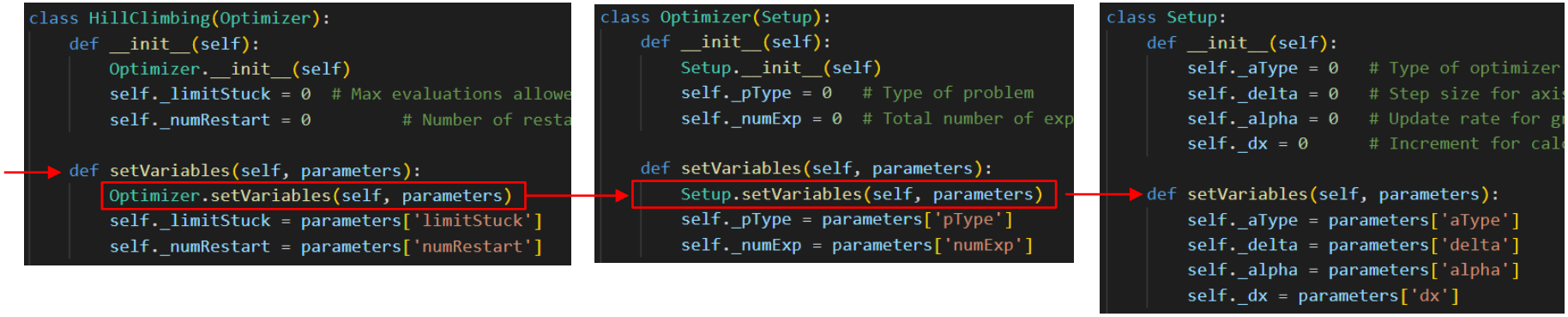
MetaHeuristics
limitEval
whenBestFound

The 4 solution algorithms (SA, FC, Stochastic, GD)

Simulated Annealing and GA

Adding More Algorithms and Classes

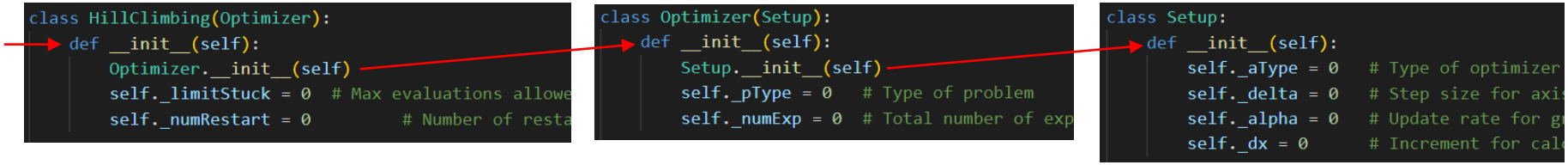
- In most cases, you create and use an instance of the class located at the lowest level of the class inheritance relationship.
  - Problem Type: Numeric / Tsp
  - Solution Algorithms: HC(SA, FC, Stochastic, GC), MetaHeuristics(SimulAnn, GA)
- This way, you can use the member variables and methods defined in the **superclass.**
  - Specifically, when inheriting and using methods with the same name from multiple superclasses, you must **explicitly specify** which class's method to use.
    - For example, the `HillClimbing` class's `setVariables` method calls the `Optimizer` class's `setVariables` method.
    - Example call: **`Optimizer.setVariables(self, parameters)`**



Adding More Algorithms and Classes

- In most cases, you create and use an instance of the class located at the lowest level of the class inheritance relationship.
  - Problem Type: Numeric / Tsp
  - Solution Algorithms: HC(SA, FC, Stochastic, GC), MetaHeuristics(SimulAnn, GA)
- This way, you can use the member variables and methods defined in the **superclass.**

  - When creating a class instance, the related **initializer methods** (*__init__*) must be called so that all variables, up to the topmost superclass, are appropriately initialized or set.



Source Code Modification Summary

Code	Change details
`setup.py`	• The ***aType*** variable was added, and the ***setVariables*** method was added.
`problem.py`	• There are no major changes compared to the existing v3 code, but there are some minor changes, such as adding member variables to store algorithm execution results, and changes to functions that output execution results.
`main.py`	• The way information like **problem type/algorithm type** is passed has changed. There is a major change in the user interface, such as reading variables (e.g., *dx*) that were previously fixed as default values from a text file, leading to overall code modification.
`optimizer.py`	• A new superclass called ***Optimizer*** was added. • Due to the addition of 3 solution algorithms (*random*, *stochastic*, *simulated annealing*), new classes were defined: ***Stochastic***, ***MetaHeuristics***, ***SimulatedAnnealing***.

Source Code Structure

- Key methods by source code file (focusing on newly added methods)
- `main.py`

Method	Description
`readPlanAndCreate`, `readValidPlan`, `readPlan`	• Obtains information related to the problem and algorithm execution from the file (at this time, lines treated as comments in the file are skipped). • Creates separate **class instances** corresponding to the problem and the algorithm.
`conductExperiment`	• Executes the algorithm to **find a solution** for the given problem.

Source Code Structure

- Key methods by source code file (focusing on newly added methods)
- `optimizer.py`

Method	Description
`HillClimbing.randomRestart`	• Executes the **run** method for *numRestart* times and saves the best solution among them (*storeResult*) • **Note:** The *run* method starts by generating a **random initial solution** and repeats until a better solution no longer emerges.
`Stochastic.run`	• Similar to *steepest ascent*, it generates **neighbor solutions** randomly, then **probabilistically selects** the best solution among them (*stochasticBest*).
`Stochastic.stochasticBest`	• **Probabilistically selects one** from among the neighbor solutions (at this time, it uses the results of evaluating each solution).

Source Code Structure

- Key methods by source code file (focusing on newly added methods)
- `optimizer.py`

Method	Description
`SimulatedAnnealing.run`	• Generates **one random neighbor solution** (similar to *FirstChoice*). • Selects the neighbor solution **probabilistically**.
`SimulatedAnnealing.initTemp`	• Sets the initial parameters so that the **probability of selecting a bad neighbor solution is 0.5** (Initial temperature setting).
`SimulatedAnnealing.tSchedule`	• **Gradually decreases the probability** of selecting a bad neighbor solution as the *iteration* repeats. • That is, it is gradually decreased by the temperature value (*temperature* is high -> high probability of selecting a bad solution).