

Search Algo. (= 탐색) is the foundation of AI algorithm!

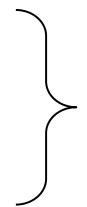
# Search Algorithms: Object-Oriented Implementation (Part A)

In Machine Learning and Deep Learning problems, we perform optimization, and search techniques are used for this. From this lecture onward, we will study the theory of basic search algorithms and how to implement them using OOP.



# Contents

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



This Part A cover the following topics

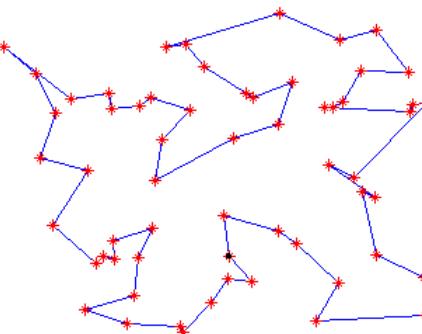
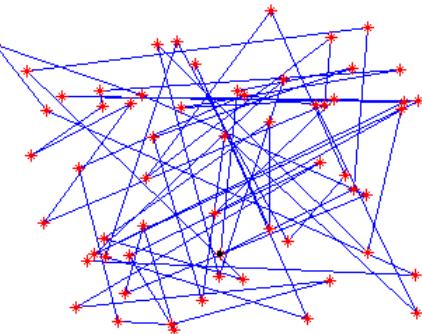
# Conventional vs. AI Algorithms

- **Intractable problems** (Problems that are difficult to handle in a mathematical sense...)
  - There are many optimization problems (최적화 문제) that require a lot of time to solve, but no efficient algorithms have been identified
  - Exponential (time complexity) algorithms are useless except for very small problems

Example. Traveling Salesperson Problem (외판원 문제)

The salesperson wants to minimize the total traveling cost (예 : 이동 시간 / 거리) required to visit all the cities in a territory, and return to the starting

ing



Visits  $n$  cities randomly without a plan  
→ Travel path is longer (= higher cost)

Optimal travel path

# Conventional vs. AI Algorithms

- Combinatorial Explosion: there are  $n!$  routes to investigate
  - Efficiency is the issue! (= Time to calculate the optimal path and order among all routes is too long.)
    - Time-efficient algorithms designed to find an approximate solution
- Nearest neighbor heuristic algo.: always goes to the nearest city
  - Given a starting city, there are  $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$  cases to consider at most
    - Starting from a city, visit the nearest among the remaining  $n-1$  cities, then the nearest of  $n-2$ , and so on....
  - Since there are  $n$  different ways to start, the total number of cases is  $n \times n(n - 1)/2$ , which is much less than  $n!$
  - One can find a near-optimal (최적 성능에 근접한) soln. in a much shorter time
- Conventional algorithms (e.g., sorting algorithms) are often called exact algorithms because they always find a correct or an optimal solution, but they can be inefficient in time
- AI algorithms use heuristics or randomized strategies to find a near optimal solution quickly

# Local Search Algorithms

- Local Search Algorithm is a category of search methods.
  - Not by checking **all** cases at once, but by iteratively improving the solution through **local search** of the neighborhood.
- Iterative improvement algorithms: (Algorithms that iteratively improve the solution)
  - State space := set of “complete” configurations/solutions
    - A complete configuration : Arbitrary solution that satisfies the conditions.
      - ✓ An arbitrary solution is valid, but not necessarily optimal (best).
      - ✓ Ex: In TSP, any tour visiting all cities (satisfies conditions) is a complete configuration.
    - State space : set of complete configuration
      - ✓ Ex: in TSP, complete tour = state space
  - Among state space, find **optimal configuration** according to the **objective function** (목적 함수)
    - Ex: Optimal complete tour minimizing total path length (Objective/Goal function)?
  - Configurations should satisfy **constraints**
    - Constraints may exist (Ex: Must go via C to visit A).
  - **Start with a complete configuration** and make modifications to improve its quality
    - Local Search starts with an arbitrary config, selects a neighbor, and iteratively improves (to reduce complexity)

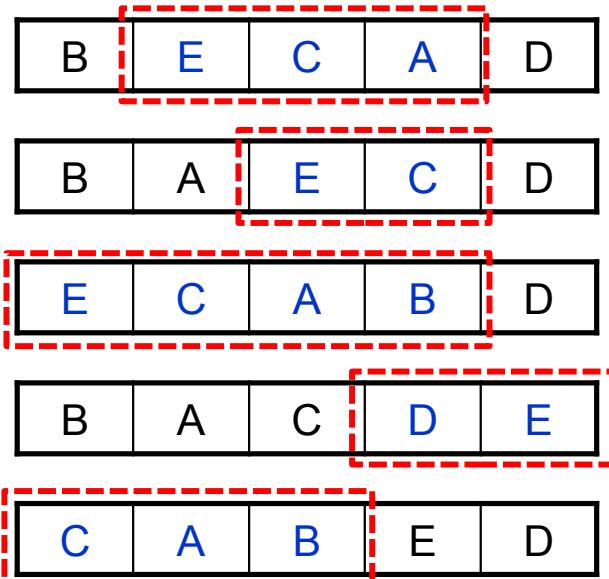
CAUTION! Search aims for the optimal solution among many/infinite complete configurations. A "solution" is just a valid state, not necessarily the most optimal one.

# Local Search Algorithms

Example: TSP (Visiting 5 cities, A-B-C-D-E)

- [Step 1] Current configuration: (Select an arbitrary complete tour and calculate cost.)  

|   |   |   |   |   |
|---|---|---|---|---|
| B | A | C | E | D |
|---|---|---|---|---|
- [Step 2] Find a slightly different (neighbor) configuration, and if valid, select the one with the minimum cost.
  - o Candidate neighborhood configurations (of the **current** configuration):



**Mutation by inversion:** // One of the techniques for selecting candidates (neighbors)

- A random subsequence is inverted (Select a random range and reverse the values within that range.)
- Select the best complete tour among candidates and repeat the inversion process to gradually find a better tour.

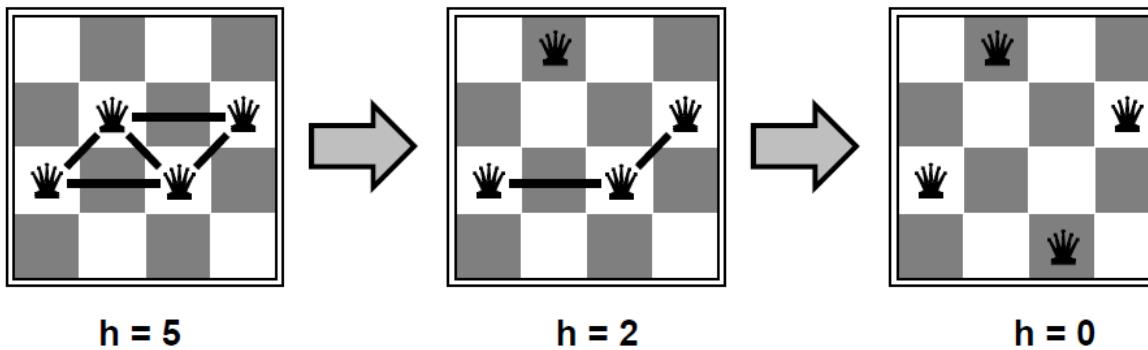
Performance:

Variants of this approach get within 1% of optimal very quickly with thousands of cities

# Local Search Algorithms

## Example: $n$ -queens

- Move a queen each time to reduce number of conflicts (서로 공격 또는 대치하는 queen의 쌍을 최소화 하기위해, 퀸의 위치를 하나씩 이동시키기)
  - Queens connected by a solid line are in an attacking position. ( $h = \# \text{lines} = \# \text{conflicts}$ )



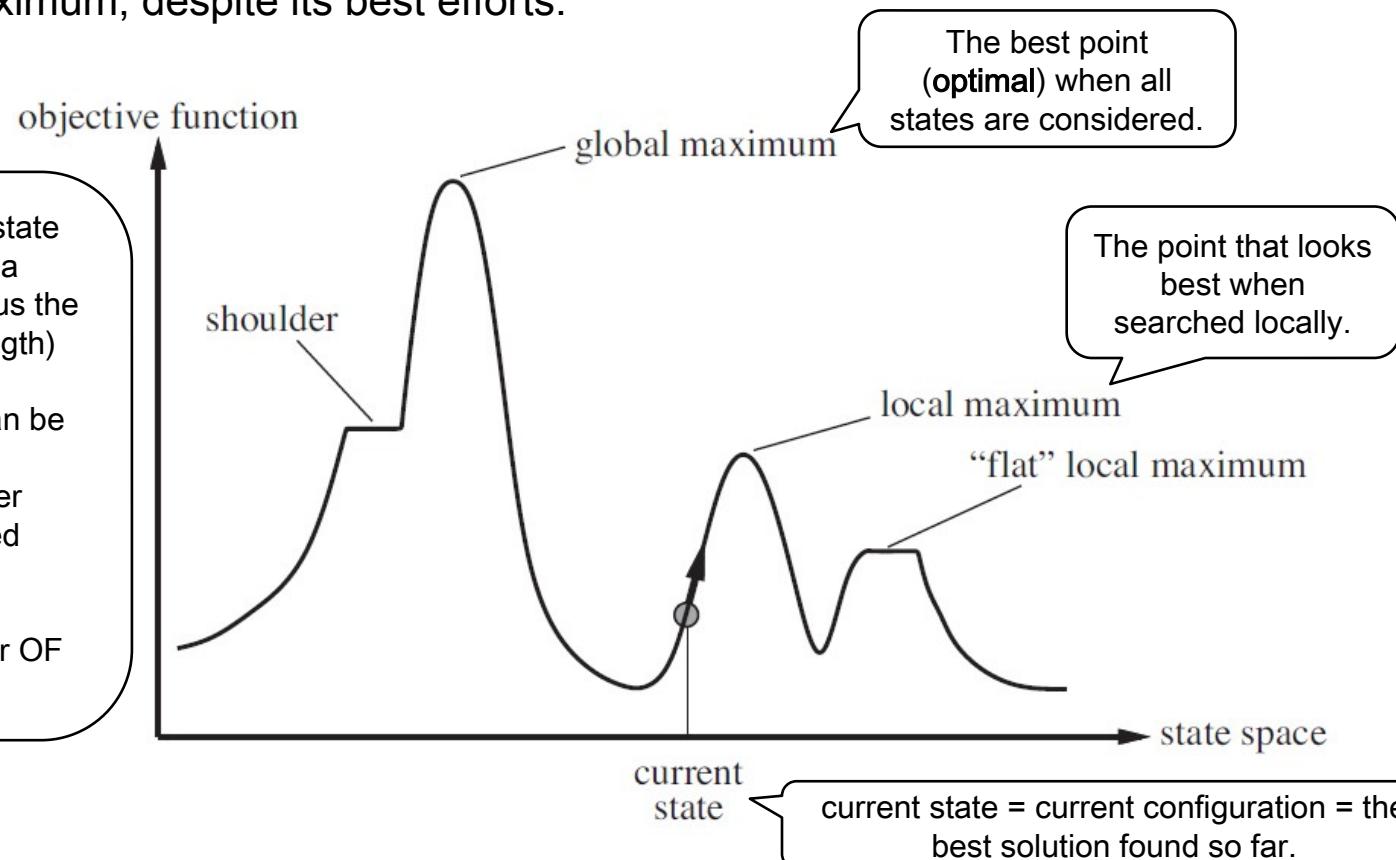
- Applying the Local Search algorithm allows us to gradually move the queens to minimize conflicts.
  - Almost always solves  $n$ -queens problems almost instantaneously (= Good time efficiency) for very large  $n$ , e.g.,  $n = 1 \text{ million}$

# Local Search Algorithms

- **State space landscape** (The overall shape of the state space distribution)

- x-axis: state (= configuration = solution)
- y-axis: objective function (OF) or cost function
- CAUTION! Local search generally reaches a local maximum, not the **optimal** global maximum, despite its best efforts.

CAUTION! A "solution" is just a valid state satisfying conditions. A problem can have many solutions, but finding the optimal one is key.



# Hill-Climbing Search

- “Like climbing Everest in thick fog with amnesia”
  - (Fog) → Cannot see far (i.e., doesn't consider all possibilities) and only inspects the vicinity (local search) → Limits search scope, reducing computational complexity.
  - (Amnesia) → Does not remember the path taken so far → Reduces memory usage.
  - Continually moves in the **increasing direction**
    - A strategy of repeatedly moving in the direction of increasing altitude toward the mountain peak.
    - The strategy of slightly changing the configuration (solution) in a direction that improves the target value
      - ✓ Ex: The strategy of slightly adjusting a stock portfolio to maximize profit.
      - ✓ Ex: On the graph  $y = -x^2$ , the strategy of moving  $x$  slightly left or right to maximize  $y$  value.
      - ✓ Note: For some problems, it may be better to move in the direction of *lowering* the target value (Ex: Gradually changing city order in TSP to minimize total path length)



# Hill-Climbing Search

- “Like climbing Everest in thick fog with amnesia”
  - Also called gradient ascent/descent search (in case of continuous space)
    - Maximize value using ascent, minimize value using descent.
    - However, for minimization problems, you can solve it as a maximization problem by setting the goal value = goal value \* -1

[Steepest **ascent** version]

Details on next page...

```
1: function HILL-CLIMBING(problem) returns a state that is a local maximum
2:
3: current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
4: loop do
5:   neighbor  $\leftarrow$  a highest-valued successor of current
6:   if neighbor.VALUE  $\leq$  current.VALUE then return current.STATE
7:   else current  $\leftarrow$  neighbor
```

# Hill-Climbing Search

- “Like climbing Everest in thick fog with amnesia”  
[Steepest **ascent** version]

**1: function HILL-CLIMBING(*problem*) returns a state that is a local maximum**

HILL-CLIMBING function that takes the problem as input (Ex: moving to the mountain peak) and returns the best result (**local maximum**)

**3: *current*  $\leftarrow$  MAKE-NODE(*problem.INITIAL-STATE*)**

Uses the initial state as input (Ex: location of the trail entrance) to create the *current* state  
(current position = an arbitrary solution)

**4: loop do**

Repeats until the best result(local max) is found

Among algorithms that gradually improve a solution, one solution must be given to begin improvement, so an initial solution is randomly selected.

**5: *neighbor*  $\leftarrow$  a highest-valued successor of *current***

Selects the highest-valued state among the neighbors (*neighbor, successor*) of the *current* state  
(Ex: Select the highest elevation point among the spots immediately accessible from the trail entrance).

**6: if *neighbor.VALUE*  $\leq$  *current.VALUE* then return *current.STATE***

If the selected *neighbor* is not better than the current state (Ex: if elevation is lower), it assumes no better solution exists and returns the current state.

**7: else *current*  $\leftarrow$  *neighbor***

If the *neighbor* is better than the current state (Ex: elevation is higher), move to that point, save it as *current*, and repeat the loop.

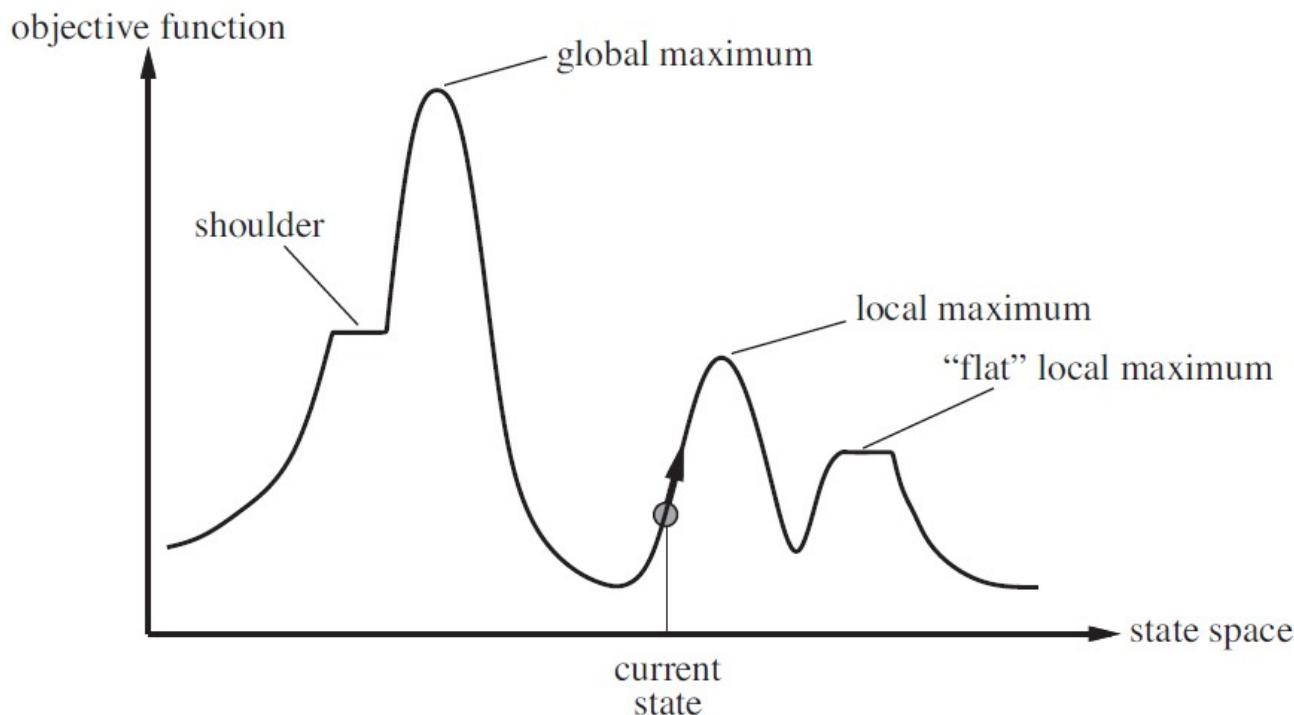
# Hill-Climbing Search

Example: Finding optimal locations for 3 airports in Romania

- state space: x,y coordinates of 3 candidate airport locations  
 $(x_1, y_1), (x_2, y_2), (x_3, y_3)$
- Objective function:  
 $f(x_1, y_1, x_2, y_2, x_3, y_3) = \text{Sum of distances from all cities to the nearest airport} \rightarrow$   
Minimize  $f$  (minimization problem)
- Hill-Climbing Search Method
  - o Randomly select initial locations for the 3 airports
  - o Slightly move airport locations to select candidate areas (= find neighbor/adjacent solutions)  $\rightarrow$  Check if objective function increases or decreases  $\rightarrow$  Repeat process of moving to better airport location

# Hill-Climbing Search

- Drawback: often gets stuck to **local maxima** due to greediness
  - Moves to the seemingly best state by only considering nearby neighboring points (= greedy approach)
  - Applying Hill-Climbing to the graph below, it immediately returns after reaching a local maximum (no better solutions nearby), failing to reach the actual optimal point (global maximum)



# Hill-Climbing Search

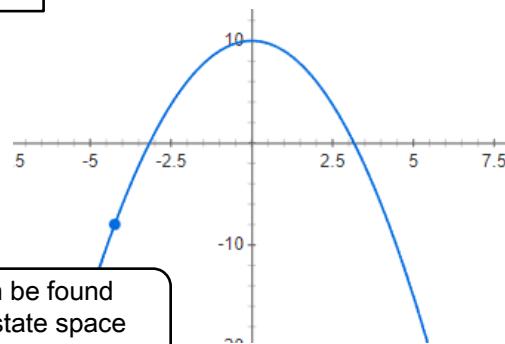
- Drawback: often gets stuck to **local maxima** due to greediness
- Possible solutions (Ways to overcome the drawback of getting stuck at *local max*):
  - key idea: Don't necessarily choose a point just because it looks good immediately!
  - **Stochastic hill climbing:**
    - Chooses at random among 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.
  - **First-choice (simple) hill climbing:**
    - Generates successors randomly until one is found that is better than the current state
    - Selects candidate states randomly and chooses the first one that is better than the current state (simple calculation)
  - **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
  - Note: Even using the above techniques may not guarantee to find the global maximum.

# Hill-Climbing Search

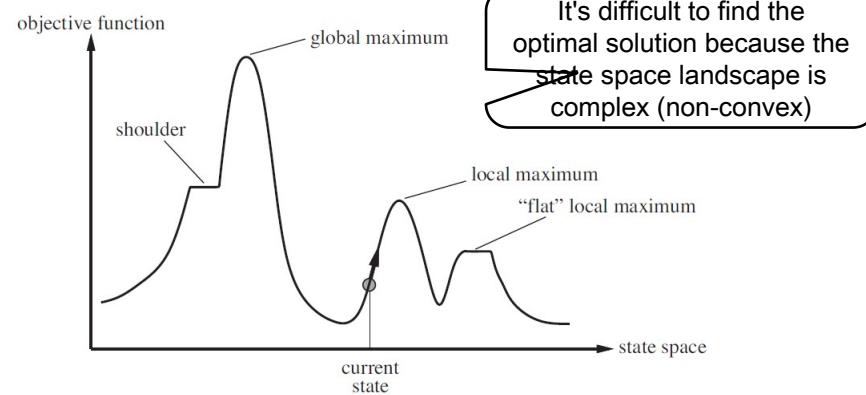
- Complexity (시간 복잡도, time complexity):
  - A standard for evaluating the time required to execute an algorithm (specifically, counting the number of basic operations needed to complete the algorithm)
  - The success of hill climbing depends on the shape of the state-space landscape
  - NP-hard problems typically have an exponential number of local maxima to get stuck on
  - A reasonably good local maximum can often be found after a small number of restarts with different starting point

Note: The concept of space complexity is also present

Reason why Local search techniques are widely used



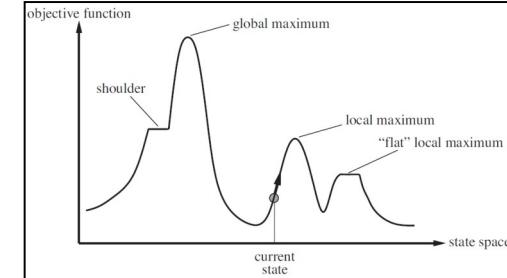
Since the final result may vary depending on the randomly selected initial state, calculate optimal solutions for various starting points and select the best one among them



# Continuous State Spaces

- Terminology Summary

- Solution: A single answer (complete configuration) that solves the given problem
  - Ex: In a problem planning a route from Seoul visiting (Chuncheon/C, Daegu/D, Busan/B), one solution is Seoul > Busan > Chuncheon > Daegu
- Optimal Solution: A solution that maximizes or minimizes the objective function
  - Ex: In the route planning problem above, the optimal solution minimizes the total travel distance (e.g., Seoul > Chuncheon > Daegu > Busan)
- state space (SS): The set of all solutions that satisfy the given constraints
  - Ex: In the route planning problem above, the set of all possible combinations like S>C>D>B, S>D>B>C, S>B>C>D, ...
- Optimal Solution (Cont.) // Refer to the top right figure
  - Global** Optimal Solution: The best choice among all options within the SS set
  - Local** Optimal Solution: The best among states adjacent to the current state; a state where no better-looking solution can be found by moving into a neighboring state
- discrete** state space : state space가 discrete (이산)이다
  - The number of elements (= number of solutions) in the SS set can be counted
  - Ex: given  $x=\{1,2,3,4,5\}$ , what is the solution that maximizes  $x^2$ ? Number of elements in SS set: 5
  - Ex: with {서울(start),춘천,대구,부산}, what is the optimal route? Number of elements in SS set:  $3 \times 2 \times 1$
- Continuous** state space : state space 가 continuous (연속적)이다
  - The number of elements in the SS set cannot be counted.
  - Ex: given range of  $t=3 \sim 31$  what is the solution that minimizes  $x^2$ ? The number of selectable x values



# Continuous State Spaces & Gradient Method

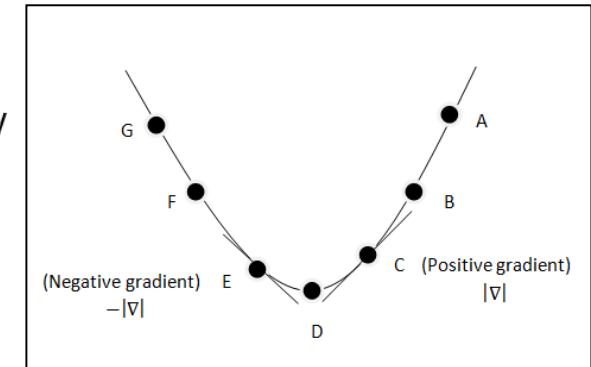
- Gradient (미분, 기울기) method

- Given an objective function  $f$ , gradient ( $\nabla f$ ) indicates the direction that increases the objective function

$$\frac{dy}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

- The Gradient method incrementally updates the state/solution ( $= x$ ) in the direction that increases the objective function  $f$ , using the gradient

- If the Gradient = 0, it is assumed the optimal solution has been found, as there is no longer a direction to increase the objective function (Note: The optimal solution found may not always be the global optimal solution)
    - It is an effective technique for finding the  $x$  that maximizes the objective function  $f$  in a continuous state space



- Comparison: Hill-Climbing must select the best state among neighboring states based on the current state ( $= x$ ), while the gradient method uses the gradient to find the best neighboring state in one step.

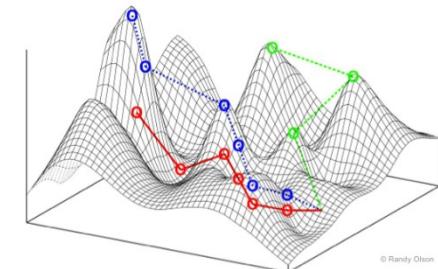
# Continuous State Spaces & Gradient Method

- Gradient (미분, 기울기) methods attempt to use the gradient of the landscape to maximize/minimize  $f$  by the following update rule:

$$\mathbf{x} \leftarrow \mathbf{x} +/\!-\alpha \nabla f(\mathbf{x}) \quad (\alpha : \text{update rate})$$

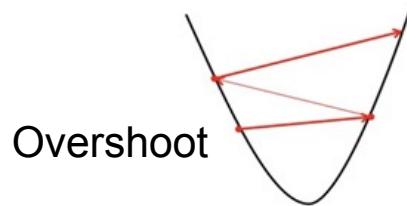
Rule for incrementally updating state  $x$

where  $\nabla f(\mathbf{x})$  is the gradient vector (containing all of the partial derivatives) of  $f$  that gives the magnitude and direction of the **steepest slope**

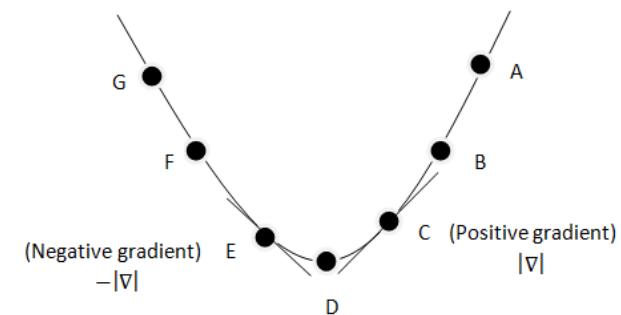


© Randy Olson

- In this case,  $\alpha$  (step=size) must be chosen carefully
  - Too small  $\alpha$ : too many steps are needed
  - Too large  $\alpha$ : the search could **overshoot** the target
  - Points where  $\nabla f(\mathbf{x}) = 0$  are known as critical points



If reached, no better state can be found nearby



# Continuous State Spaces & Gradient Method

Example: Gradient **descent** (Goal is to minimize the objective function)

- If  $f(x) = x^2 + 1$ , then  $f'(x) = 2x$

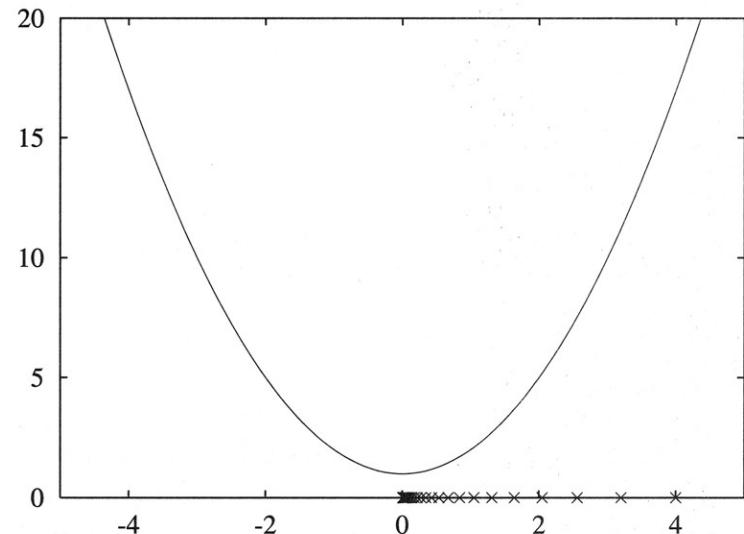
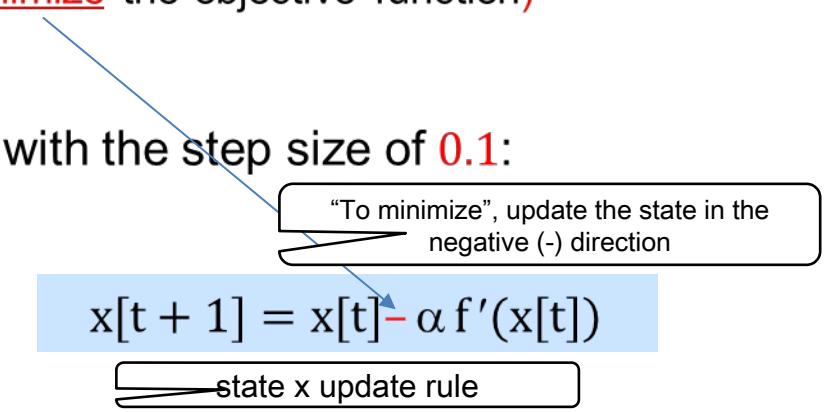
Starting from an initial value  $w = 4$ , with the step size of **0.1**:

- o  $4 - (0.1 \times 2 \times 4) = 3.2$
- o  $3.2 - (0.1 \times 2 \times 3.2) = 2.56$
- o  $2.56 - (0.1 \times 2 \times 2.56) = 2.048$

.....

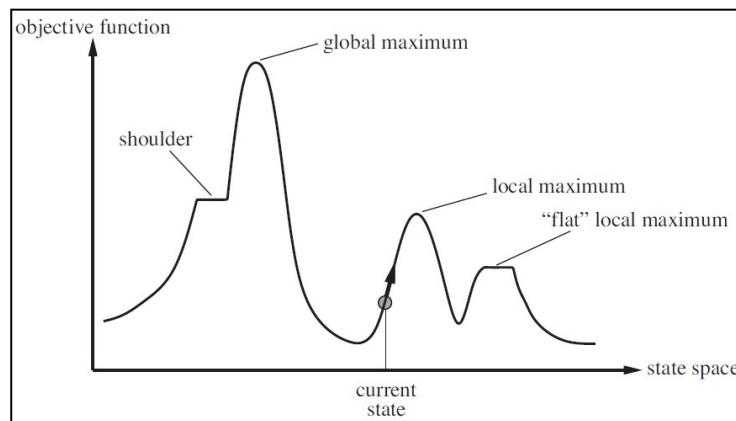
- Stops when the change in parameter value becomes too small
  - o Ex: stop if  $|x[t+1] - x[t]| \leq 1e-5$
  - o or, stop if  $|f'(x[t])| \leq 1e-5$

Because reaching the Critical point *exactly* is probabilistically almost impossible



# Simulated Annealing Search

- Idea:
  - Hill-Climbing/Gradient method examines the surroundings and makes the best choice from them (i.e., makes a locally optimal choice)
    - HC : Selects the best state among neighboring states
    - Grad. : Calculates the derivative (gradient) based on the current position, and moves in that direction to select a new state
    - Both techniques stop when there is no better state
  - However, if the starting position is poor, it may reach and terminate at a local optimal solution.
  - Furthermore, always moving in the seemingly best direction after only checking the vicinity may be a poor decision



# Simulated Annealing Search

- Idea:
  - Efficiency of **valley-descending** + completeness of random walk
    - That is, select a better next state in a time-efficient manner, but add a "random" selection process when choosing the next state
  - Escape local **minima** by allowing some “bad” moves. **But gradually decrease their step size and frequency**
    - Randomly allows bad moves, which helps to find better solutions
    - However, bad moves should be performed restrictively (i.e., gradually reduce the frequency of bad moves and the extent to which they change the state)

# Simulated Annealing Search

- Origin of the Algorithm's Name (annealing: metal tempering/heat treatment)
  - Originates from the method of heating a metal material and then slowly cooling it (changing the temperature) to eliminate internal defects and encourage optimal physical properties (i.e., cool quickly at first, then slowly later to achieve stable properties!)
    - Devised by Metropolis *et al.*, 1953, for physical process modeling
    - Widely used in VLSI layout, airline scheduling, etc.
  - At fixed temperature  $T$ , state occupation probability reaches Boltzmann distribution  $p(x) = \alpha e^{-E(x)/kT}$
  - $T$  decreased slowly enough → always reach the best state

# Simulated Annealing Search

[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*  $\leftarrow$  MAKE-NODE(*problem*.INITIAL-STATE)

Calculate the current state from the initial state

**for**  $t \leftarrow 1$  **to**  $\infty$  **do**

$T \leftarrow \text{schedule}[t]$

**if**  $T = 0$  **then return** *current*

*next*  $\leftarrow$  a randomly selected successor of *current*

Randomly select the next state

$\Delta E \leftarrow \text{next.VALUE} - \text{current.VALUE}$

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

**if**  $\Delta E < 0$  **then** *current*  $\leftarrow$  *next*

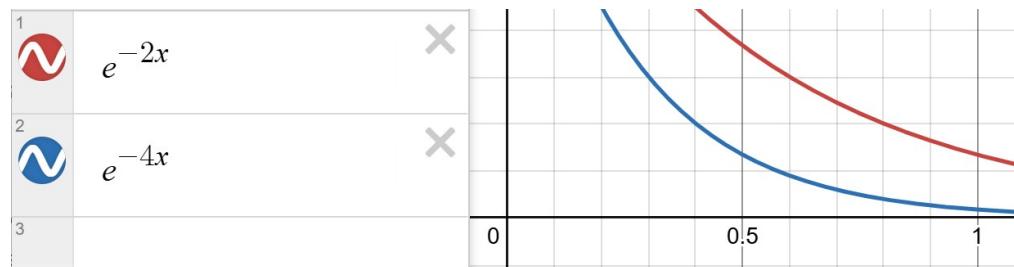
Assumes a minimization problem  $\rightarrow$  Lower value is better

**else** *current*  $\leftarrow$  *next* only with probability  $e^{-\Delta E/T}$

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

# Simulated Annealing Search

- A **random** move is picked (with decreasing probability) instead of the best move
- If the move improves the situation, it is always accepted
- Otherwise, the move is accepted with probability  $e^{-\Delta E / T}$ 
  - $\Delta E$ : the amount by which the evaluation is worsened
    - The acceptance probability decreases exponentially with the “badness” of the move
    - i.e., The worse the next state is, the lower the probability of selecting it.
  - $T$ : temperature, determined by the **annealing schedule** (controls the randomness)
    - Bad moves are more likely at the start when  $T$  is high
      - ✓ When the initial  $T$  value is high, the probability of accepting a random bad move is high.
    - They become less likely as  $T$  decreases
      - ✓ Since it's common to set the schedule so that  $T$  decreases as the iteration progresses, the probability of selecting a bad move decreases with each iteration.
  - In short, random moves are highly permitted in the early stages, and restricted in the later stages



# Simulated Annealing Search

- $T \rightarrow 0$ : simple hill-climbing (first-choice hill-climbing)
  - If  $T$  approaches 0, the probability of accepting any move that worsens the solution ( $\Delta E > 0$ ) drops to zero. The algorithm is then forced to only accept improving moves ( $\Delta E < 0$ ), which is the defining characteristic of a greedy Hill-Climbing search.
- If the annealing schedule reduces  $T$  slowly enough, a global optimum will be found with probability approaching 1
  - In practice, following this theoretically slow schedule would require an **impractically large number of iterations** (often exponential time)!
- The initial temperature is often heuristically set to a value so that the probability of accepting bad moves is 0.5
  - This high initial probability ensures the algorithm is highly exploratory at the start.

# Search Algorithms: Object-Oriented Implementation (Part B)

구현 시작 !

# Contents

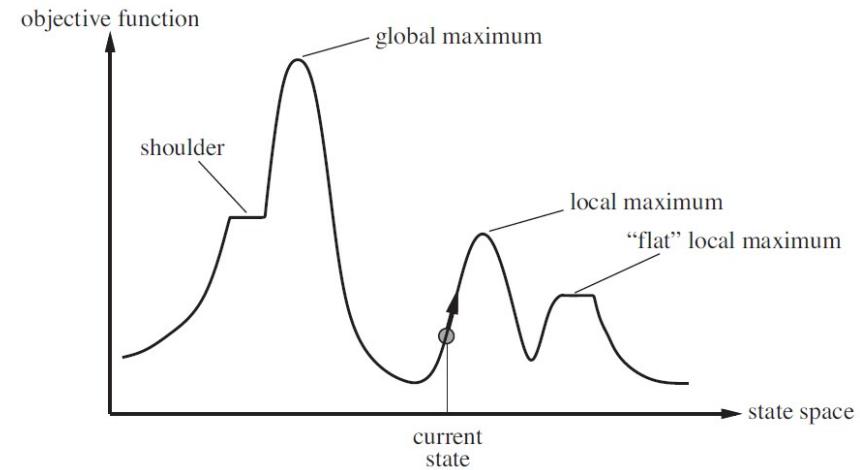
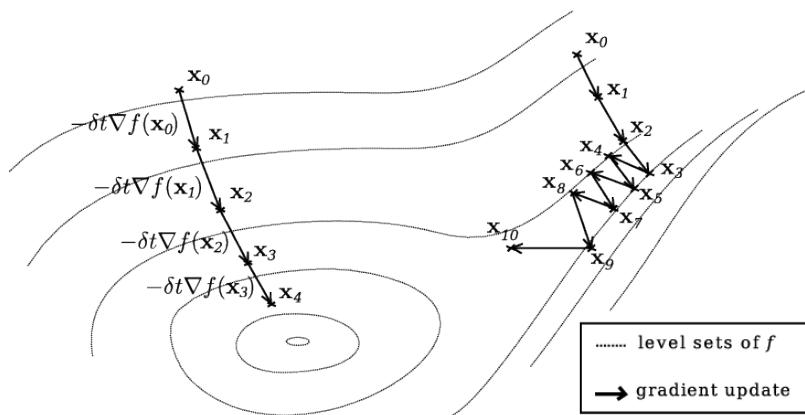
- Conventional vs. AI Algorithms
- Local Search Algorithms
- Implementing Hill-Climbing Algorithms
- Defining ‘Problem’ Class
- Adding Gradient Descent
- Defining ‘HillClimbing’ Class
- Adding More Algorithms and Classes
- Adding Genetic Algorithm
- Experiments

앞으로는, **Search Algo** 을 구현하는 과제가  
진행되고, 과제에서 구현해야 하는 코드의  
일부를 수업시간에 살펴보는 방식으로  
진행합니다.



# Implementation Reference Notes

- Search Algorithm
  - The techniques/algorithms covered in the class are methods for **iteratively finding the optimal solution**, and are called **Search Algorithms**.
  - However, the optimal solution found by the algorithm can be **global optimal** or **local optimal**.



# Implementation Reference Notes

- Search Tool
  - In order to **execute a Search Algo.** (Algorithm), tasks such as **problem generation/loading and parameter setting** must be performed. // **Preprocessing** (or 'pre-work')
  - Additionally, a function to **display the Search Algo. execution results** is also necessary. // **Post-processing** (or 'post-work')
  - A program that performs the above steps: 1) Preprocessing, 2) Algorithm execution, and 3) Post-processing is called a **Search Tool**.

# Implementation Reference Notes

- Search Tool Version
  - The Search Tool is implemented in various versions (iteratively expanded & improved).
    - v0 ( 강의 . Search Algo. B) : A version where **all the code to implement one algorithm is contained in a single file**
    - v1 ( 강의 . Search Algo. B) : A version implemented by **modularizing the logic common to different pieces of code into a separate file**, then importing and using that module/file (code refactoring & reuse)
    - v2 ( 강의 . Search Algo. C) : A version implemented using **classes** (v2.1, v2.2)
    - v3~v5 : Versions implemented using classes, with some algorithms added, leading to **iterative improvements in the code structure**
      - v3 : 강의 . Search Algo. D
      - v4 : 강의 . Search Algo. E
      - v5 : 강의 . Search Algo. F

Today's

# Implementing Hill-Climbing Algorithms

- Our eventual goal is to implement an **optimization tool** that can run various search algorithms
- We begin to implement **two hill climbing algorithms** each **for two different types of problems**: (we implement a total of **four** hill climbing algorithms)
  - Steepest-ascent hill climbing **for numerical optimization** (**SAHC-N**)
  - First-choice hill climbing **for numerical optimization** (**FCHC-N**)
  - Steepest-ascent hill climbing **for TSP** (**SAHC-T**)
  - First-choice hill climbing **for TSP** (**FCHC-T**)

Ref:

- numerical optimization : The task of finding the variables that **maximize or minimize** the objective function/value given a mathematical expression/objective function. // **continuous**
- TSP : The problem of calculating the shortest distance to visit all given cities and return to the starting city. // **discrete**

복습 : Hill Climbing Algo. ( 왼쪽 ) vs first-choice hill climbing ( 오른쪽 )

[Steepest ascent version]

```
1: function HILL-CLIMBING(problem) returns a state that is a local maximum  
2:  
3: current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)  
4: loop do  
5:   neighbor  $\leftarrow$  a highest-valued successor of current  
6:   if neighbor.VALUE  $\leq$  current.VALUE then return current.STATE  
7:   else current  $\leftarrow$  neighbor
```

– First-choice (simple) hill climbing:

- Generates successors randomly until one is found that is better than the current state
- 후보 state를 무작위로 선택하고, current state 보다 좋다면 선택(계산이 간단)

- If the goal is to maximize (\$\maximize\$) the objective function, and there is no better solution (neighbor):
- XXX.VALUE: Refers to the **calculated value of the objective function** for the solution XXX.
- Example: If objective function( $x$ ) =  $x^2$ , current( $x = 4$ ), then current.VALUE =  $(4)^2$ .

Today, we'll only looked at how to implement the two problems of the **Numerical Optimization** type.



Implementation by including all necessary logic in a single source code file

# SEARCH TOOL V0

# Numerical Optimization

- The problem of **finding the variable values to minimize or maximize a given mathematical expression.**
- Assume the problem is input via a **text file**, as shown below.
  - The problem is to find the variable values that **minimize the mathematical expression** entered on the first line
  - Each variable has a minimum and maximum value range.
    - For example, the variables  $x_1 \sim x_5$  below can only select values in the range of -30 to +30.

```
Convex.txt
1 (x1 - 2) ** 2 + 5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2
2 x1,-30,30
3 x2,-30,30
4 x3,-30,30
5 x4,-30,30
6 x5,-30,30
```

# Numerical Optimization

- Execution Results (Example using the **Steepest-Ascent Algorithm**)

```
Enter the file name of a function: problem/Convex.txt
```

Input the name and path of the .txt file defining the problem (User Input).

Objective function:

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

Search space:

$x_1: (-30.0, 30.0)$   
 $x_2: (-30.0, 30.0)$   
 $x_3: (-30.0, 30.0)$   
 $x_4: (-30.0, 30.0)$   
 $x_5: (-30.0, 30.0)$

- \* **Variable and Range:** The value of each variable can be changed within its specified range.
- \* **Problem Goal:** Find the  $x_1, \dots, x_5$  values that make the **value of the objective function the smallest**.

Search algorithm: **Steepest-Ascent** Hill Climbing

Mutation step size: 0.01

The calculated optimal solution

Solution found: (1.006, 5.002, 7.007, 1.005, 7.002)

Minimum value: 0.000

The result when the optimal solution is substituted into the objective function.

Total number of evaluations: 95,321

# Numerical Optimization

- Execution Results (Example using the **First-Choice Algorithm**)

```
Enter the file name of a function: problem/Convex.txt  
Input the name and path of the .txt file defining the problem (User Input).  
  
Objective function: (x1 - 2) ** 2 + 5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 - 7) ** 2  
Value goal  
  
Search space:  
x1: (-30.0, 30.0)  
x2: (-30.0, 30.0)  
x3: (-30.0, 30.0)  
x4: (-30.0, 30.0)  
x5: (-30.0, 30.0)  
Main changes  
• * Variable and Range: The value of each variable can be changed within its specified range.  
• * Problem Goal: Find the  $x_1, \dots, x_5$  values that make the value of the objective function the smallest.  
  
Search algorithm: First-Choice Hill Climbing  
  
Mutation step size: 0.01  
Max evaluations with no improvement: 100 iterations  
The calculated optimal solution  
Solution found: (2.00, 5.00, -8.00, -1.00, 7.00)  
Minimum value: 0.000  
The result when the optimal solution is substituted into the objective function.  
  
Total number of evaluations: 37,336
```

# TSP Problem

- The problem of planning the **route (= order of visits)** to visit all cities with the **shortest total distance**.
- Assume the problem is input via a **text file**, as shown below.
  - The **number of cities** is on the first line.
  - The subsequent lines contain the **coordinates of each city**.

| tsp30.txt |           |
|-----------|-----------|
| 1         | 30        |
| 2         | (8, 31)   |
| 3         | (54, 97)  |
| 4         | (50, 50)  |
| 5         | (65, 16)  |
| 6         | (70, 47)  |
| 7         | (25, 100) |
| 8         | (55, 74)  |
| 9         | (77, 87)  |
| 10        | (6, 46)   |

# TSP Problem

- Execution Results (Example using the **First-Choice Algorithm**)

```
Enter the file name of a TSP: problem/tsp30.txt
```

Input the name and path of the .txt file defining the problem (User Input).

Number of cities: 30

City locations:

|           |           |          |          |          |
|-----------|-----------|----------|----------|----------|
| (8, 31)   | (54, 97)  | (50, 50) | (65, 16) | (70, 44) |
| (25, 100) | (55, 74)  | (77, 87) | (6, 46)  | (70, 78) |
| (13, 38)  | (100, 32) | (26, 35) | (55, 16) | (26, 77) |
| (17, 67)  | (40, 36)  | (38, 27) | (33, 2)  | (48, 9)  |
| (62, 20)  | (17, 92)  | (30, 2)  | (80, 75) | (32, 26) |
| (43, 79)  | (57, 49)  | (18, 24) | (96, 76) | (81, 1)  |

City starts from index 0...

The last city is City 29

Search algorithm: **First-Choice** Hill Climbing

Max evaluations with no improvement: 100 iterations

Best order of visits:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 4  | 26 | 2  | 24 | 16 | 27 | 0  | 8  | 10 | 12 |
| 17 | 18 | 22 | 19 | 3  | 13 | 20 | 29 | 11 | 28 |
| 23 | 7  | 9  | 6  | 25 | 1  | 14 | 15 | 21 | 5  |

The Optimal Order of Visits

Minimum tour cost: 491

Total number of evaluations: 677

The Total Distance traveled when following the optimal route.

# TSP Problem

- Execution Results (Example using the **Steepest-Ascent Algorithm**)

Enter the file name of a TSP: problem/tsp30.txt

Input the name and path of the .txt file defining the problem (User Input).

Number of cities: 30

City locations:

|           |           |          |          |          |
|-----------|-----------|----------|----------|----------|
| (8, 31)   | (54, 97)  | (50, 50) | (65, 16) | (70, 47) |
| (25, 100) | (55, 74)  | (77, 87) | (6, 46)  | (70, 78) |
| (13, 38)  | (100, 32) | (26, 35) | (55, 16) | (26, 77) |
| (17, 67)  | (40, 36)  | (38, 27) | (33, 2)  | (48, 9)  |
| (62, 20)  | (17, 92)  | (30, 2)  | (80, 75) | (32, 36) |
| (43, 79)  | (57, 49)  | (18, 24) | (96, 76) | (81, 39) |

Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 11 | 29 | 20 | 13 | 3  | 4  | 28 | 23 | 9  | 1  |
| 7  | 6  | 2  | 16 | 26 | 25 | 21 | 5  | 8  | 12 |
| 14 | 10 | 0  | 12 | 27 | 24 | 17 | 19 | 18 | 22 |

The Optimal Order of Visits

Minimum tour cost: 620

Total number of evaluations: 807

The **Total Distance** traveled when following the optimal route.

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **main()**:
  - Creates a problem instance by reading a function expression and its domain information from the file given by the user (`createProblem`)
    - Returns :  $p = (\text{expr}, \text{domain})$
    - `expr` : Objective Function
    - `domain` : minimization/maximization
  - Calls the search algorithm (`steepestAscent`) and obtains the results
  - Shows the specifics of the problem just solved (`describeProblem`)
  - Shows the settings of the search algorithm (`displaySettings`)
  - Reports the results (`displayResults`)

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **main()**:
  - Creates a problem instance (`createProblem`)
  - Calls the search algorithm (`steepestAscent`) and obtains the results
  - Shows the specifics of the problem just solved (`describeProblem`)
  - Shows the settings of the search algorithm (`displaySettings`)
  - Reports the results (`displayResults`)

```
def main():
    # Create an instance of numerical optimization problem
    p = createProblem()    # 'p': (expr, domain)
    # Call the search algorithm
    solution, minimum = steepestAscent(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)
```

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **createProblem()**:

- Gets a file name from the user and reads information from the file
    - Function expression – saved as a string **# problem instance**
    - Variable names – saved as a list of strings
    - Lower bounds of the variables – saved as a list of floats
    - Upper bounds of the variables – saved as a list of floats
  - Returns the problem instance as a list of expression and domain, where the domain is a list of the followings :

- Variable names
  - Lower bounds
  - Upper bounds

the problem to be solved

# domain

Prints createProblem return value  
(=problem instance)

```
# Create an instance of numerical optimization problem
p = createProblem()    # 'p': (expr, domain)
print(p[0], end="")
print(p[1][0])
print(p[1][1])
print(p[1][2])
```

```
Enter the file name of a function: problem/Convex.txt
(x1 - 2) ** 2 + 5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 -
7) ** 2
['x1', 'x2', 'x3', 'x4', 'x5'] # var
[-30.0, -30.0, -30.0, -30.0, -30.0] # lower
[ 30.0, 30.0, 30.0, 30.0, 30.0] # upper
```

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **steepestAscent (p) :**
  - Given a problem `p`, takes a random initial point (`randomInit`) as a current point and evaluates it (`evaluate`)
  - Repeats updating the current point:
    - Generate multiple neighbors (`mutants`)
    - Finds the best one (`bestOf`) among them
    - If it is better than current, update and continue  
Otherwise, stop
  - Returns the final current point and its evaluation value

- “neighbor” = Solution adjacent to `current`
- Adjacent solution = Solution obtained by slightly changing current solution

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **steepestAscent(p):**

```
def steepestAscent(p):
    current = randomInit(p) # 'current' is a list of values
    valueC = evaluate(current, p)
    while True:
        neighbors = mutants(current, p)
        successor, valueS = bestOf(neighbors, p)
        if valueS >= valueC:
            break
        else:
            current = successor
            valueC = valueS
    return current, valueC
```

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **randomInit (p):**
  - Given a problem **p**, returns a point randomly chosen within **p**'s domain

Generates a solution (point) randomly within the range that satisfies the specified minimum (lower bound) and maximum (upper bound) conditions.
- **mutants (current, p):** One of the methods for generating the adjacent (neighboring) successors of **current**
  - Returns  $2n$  neighbors of **current** ( $n$ : # of variables)
    - $i$ -th variable value **plus DELTA**  $n$  number of times
    - $i$ -th variable value **minus DELTA**  $n$  number of times
  - Each neighbor is made by copying **current**, choosing an  $i$ -th variable of **p**, and then altering its value (**mutate**) by both adding and subtracting **DELTA (d)**
    - **DELTA** is a named constant representing the step size of axis-parallel mutation

# Steepest-Ascent Hill Climbing for Numerical Optimization

- ~ **mutate(current, i, d, p):**
  - Makes a copy of `current`, alter the value of  $i$ -th variable by adding `d` as long as the new value remains within the domain of `p`, and returns the resulting mutant

```
def mutate(current, i, d, p): ## Mutate i-th of 'current' if legal
    curCopy = current[:]
    domain = p[1]          # [VarNames, low, up]
    l = domain[1][i]        # Lower bound of i-th
    u = domain[2][i]        # Upper bound of i-th
    if l <= (curCopy[i] + d) <= u:
        curCopy[i] += d
    return curCopy
```

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **evaluate(current, p):**
  - Evaluates the expression of problem  $p$  after assigning the values of `current` to its variables, and returns the result
    - Global variable `NumEval` is used to count the number of evaluations

```
def evaluate(current, p):  
    ## Evaluate the expression of 'p' after assigning  
    ## the values of 'current' to the variables  
    global NumEval  
  
    NumEval += 1  
    expr = p[0]          # p[0] is function expression  
    varNames = p[1][0]   # p[1] is domain  
    for i in range(len(varNames)):  
        assignment = varNames[i] + '=' + str(current[i])  
        exec(assignment)  
    return eval(expr)
```

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **bestOf(neighbors, p):**
  - Evaluates each candidate solution in `neighbors` (`evaluate`), identifies the best one, and returns it with its evaluation value
- **describeProblem(p):**
  - Shows the expression and the domain of problem `p`

```
def describeProblem(p):  
    print()  
    print("Objective function:")  
    print(p[0])    # Expression  
    print("Search space:")  
    varNames = p[1][0] # p[1] is domain: [VarNames, low, up]  
    low = p[1][1]  
    up = p[1][2]  
    for i in range(len(low)):  
        print(" " + varNames[i] + ":", (low[i], up[i]))
```

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **displaySetting()**:
  - Shows that the steepest-ascent hill climbing has been used as the search algorithm
  - Displays the step size (**DELTA**) of the axis-parallel mutation

```
def displaySetting():
    print()
    print("Search algorithm: Steepest-Ascent Hill Climbing")
    print()
    print("Mutation step size:", DELTA)
```

```
DELTA = 0.01    # Mutation step size
```

## Steepest-Ascent Hill Climbing for Numerical Optimization

- **displayResult(solution, minimum):**
  - Reports the result of optimization, which consists of
    - **solution** (the best solution found),
    - **minimum** (its evaluation value), and
    - the total number of evaluations
  - **solution** is transformed to a tuple (**coordinate**) before printing
- **coordinate(solution):**
  - Rounds up **solution** and returns it (to the third decimal place)

# Steepest-Ascent Hill Climbing for Numerical Optimization

- **displayResult(solution, minimum):**
- **coordinate(solution):**

```
def displayResult(solution, minimum):  
    print()  
    print("Solution found:")  
    print(coordinate(solution)) # Convert list to tuple  
    print("Minimum value: {:.3f}".format(minimum))  
    print()  
    print("Total number of evaluations: {}".format(NumEval))  
  
def coordinate(solution):  
    c = [round(value, 3) for value in solution]  
    return tuple(c) # Convert the list to a tuple
```

## Steepest-Ascent Hill Climbing for Numerical Optimization

- The program is required to import ‘random.py’ and ‘math.py’

```
import random  
import math
```

# First-Choice Hill Climbing for Numerical Optimization

- **main()**:
  - The only difference is that it calls **firstChoice** instead of **steepestAscent**

```
def main():
    # Create an instance of numerical optimization problem
    p = createProblem()    # 'p': (expr, domain)
    # Call the search algorithm
    solution, minimum = firstChoice(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)
```

# First-Choice Hill Climbing for Numerical Optimization

- **firstChoice (p) :**
  - Only one random successor is generated (**randomMutant**) to update the current solution
  - The algorithm stops if no improvement is observed for a certain consecutive number (**LIMIT\_STUCK**) of iterations assuming that the search is stuck at a local minimum
    - **LIMIT\_STUCK** is a named constant in this implementation
      - Unlike the method of generating multiple **neighbor soln** and selecting the best among them, only one random neighbor soln is selected.
      - If a worse solution appears consecutively for the number of **LIMIT\_STUCK** iterations, the program stops.
      - We assume **LIMIT\_STUCK** = 100

# First-Choice Hill Climbing for Numerical Optimization

- **firstChoice(p):**

```
def firstChoice(p):  
    current = randomInit(p)    # 'current' is a list of values  
    valueC = evaluate(current, p)  
    i = 0  
    while i < LIMIT_STUCK:  
        successor = randomMutant(current, p)  
        valueS = evaluate(successor, p)  
        if valueS < valueC:  
            current = successor  
            valueC = valueS  
            i = 0                # Reset stuck counter  
        else:  
            i += 1  
    return current, valueC
```

# First-Choice Hill Climbing for Numerical Optimization

- **randomMutant(current, p):**
  - Returns a mutant of `current`, which is made by randomly choosing a variable of `p` and then altering its value (`mutate`) by either adding or subtracting DELTA
  - The **randomMutant** function is implemented by calling the **mutate** function. When doing so, it must decide **which index's stored value to modify** ( $d = \text{DELTA} = 0.01$ ).

```
def mutate(current, i, d, p): ## Mutate i-th of 'current' if legal
    curCopy = current[:]
    domain = p[1]          # [VarNames, low, up]
    l = domain[1][i]        # Lower bound of i-th
    u = domain[2][i]        # Upper bound of i-th
    if l <= (curCopy[i] + d) <= u:
        curCopy[i] += d
    return curCopy
```

## First-Choice Hill Climbing for Numerical Optimization

- **displaySetting():**
  - Shows that the search algorithm used is first-choice hill climbing

```
def displaySetting():
    print()
    print("Search algorithm: First-Choice Hill Climbing")
    print()
    print("Mutation step size:", DELTA)
```

- The functions `createProblem`, `randomInit`, `evaluate`, `mutate`, `describeProblem`, `displayResult`, and `coordinate` are all reused without change

# Hill Climbing for TSP ( 외판원 문제 )

- For TSP, the structure of the problem and its implementation are **nearly identical** to the **NumericalOptimization** problem
  - However, the logic for generating mutant are different
  - TSP generate the mutant (adjacent solution) using mutate by inversion (refer to the previous lecture)
- inversion : Generating one adjacent solution
  - Reverse the items from index  $i$  to index  $j$

```
def inversion(current, i, j): # Perform inversion
    curCopy = current[:]
    while i < j:
        curCopy[i], curCopy[j] = curCopy[j], curCopy[i]
        i += 1
        j -= 1
    return curCopy
```

The **TSP implementation** will be examined in detail in the next lecture.



# Hill Climbing for TSP ( 외판원 문제 )

- **First-Choice (TSP)** : generating a single **mutant**

```
def randomMutant(current, p): # Apply inversion
    while True:
        i, j = sorted([random.randrange(p[0])
                      for _ in range(2)])
        if i < j:
            curCopy = inversion(current, i, j)
            break
    return curCopy
```

- Variable **p[0]**: **number of cities, N** (Read from the first line of the defined problem file).
- **random.randrange(N)**: Returns a random integer from  $0 \sim N - 1$ .
  - When the number of cities is  $N$ , each city is identified by a number from **0, 1, ..., N – 1**.
- The process:
  - **Two random numbers are generated and stored in a list, then the list is sorted.** (This ensures that the two values are in order, i.e.,  $i \leq j$ ).
  - **If  $i$  and  $j$  are different ( $i < j$ ), they are treated as valid indices**, and the function calls **inversion(current, i, j)** to generate a single mutant. (The while True: loop continues until two different indices are selected such that  $i < j$ ).

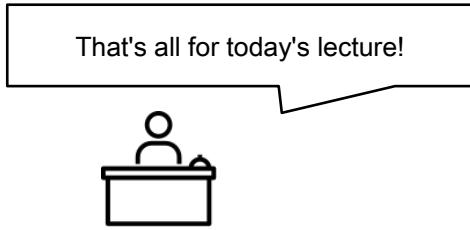
# Hill Climbing for TSP ( 외판원 문제 )

- **Steepest-Ascent (TSP)** : Create a number of **mutants**

```
def mutants(current, p): # Apply inversion
    n = p[0]
    neighbors = []
    count = 0
    triedPairs = []
    while count <= n: # Pick two random loci for inversion
        i, j = sorted([random.randrange(n) for _ in range(2)])
        if i < j and [i, j] not in triedPairs:
            triedPairs.append([i, j])
            curCopy = inversion(current, i, j)
            count += 1
            neighbors.append(curCopy)
    return neighbors
```

# Code

- Ref :
  - The **code omitted from the lecture materials must be implemented in the homework.**



That's all for today's lecture!

# Search Algorithms: Object-Oriented Implementation (Part B)

Improved implementation!

# Contents

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

# Implementation Reference Notes

- Search Tool Version
  - The Search Tool is implemented in various versions (iteratively expanded & improved).
    - v0 ( 강의 . Search Algo. B) : A version where **all the code to implement one algorithm is contained in a single file**
    - v1 ( 강의 . Search Algo. B) : A version implemented by **modularizing the logic common to different pieces of code into a separate file**, then importing and using that module/file (code refactoring & reuse)
    - v2 ( 강의 . Search Algo. C) : A version implemented using **classes** (v2.1, v2.2)
    - v3~v5 : Versions implemented using classes, with some algorithms added, leading to **iterative improvements in the code structure**
      - v3 : 강의 . Search Algo. D
      - v4 : 강의 . Search Algo. E
      - v5 : 강의 . Search Algo. F

Previous

Today's

# Implementing Hill-Climbing Algorithms

- Our eventual goal is to implement an **optimization tool** that can run various search algorithms
- We begin to implement **two hill climbing algorithms** each **for two different types of problems**: (we implement a total of **four** hill climbing algorithms)
  - Steepest-ascent** hill climbing **for numerical optimization** (**SAHC-N**)
  - First-choice** hill climbing **for numerical optimization** (**FCHC-N**)
  - Steepest-ascent** hill climbing **for TSP** (**SAHC-T**)
  - First-choice** hill climbing **for TSP** (**FCHC-T**)

Ref:

- numerical optimization : The task of finding the variables that **maximize or minimize** the objective function/value given a mathematical expression/objective function. // **continuous**
- TSP : The problem of calculating the shortest distance to visit all given cities and return to the starting city. // **discrete**

복습 : Hill Climbing Algo. ( 왼쪽 ) vs first-choice hill climbing ( 오른쪽 )

[Steepest ascent version]

```

1: function HILL-CLIMBING(problem) returns a state that is a local maximum
2:
3: current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
4: loop do
5:   neighbor  $\leftarrow$  a highest-valued successor of current
6:   if neighbor.VALUE  $\leq$  current.VALUE then return current.STATE
7:   else current  $\leftarrow$  neighbor

```

– First-choice (simple) hill climbing:

- Generates successors randomly until one is found that is better than the current state
- 후보 state를 무작위로 선택하고, current state 보다 좋다면 선택(계산이 간단)

- If the goal is to maximize the objective function, and there is no better solution (neighbor):
- XXX.VALUE**: Refers to the **calculated value of the objective function** for the solution **XXX**.
- Example**: If objective function( $x$ ) =  $x^2$ , current( $x = 4$ ), then current.VALUE =  $(4)^2$ .

Implementation with modularized common logic

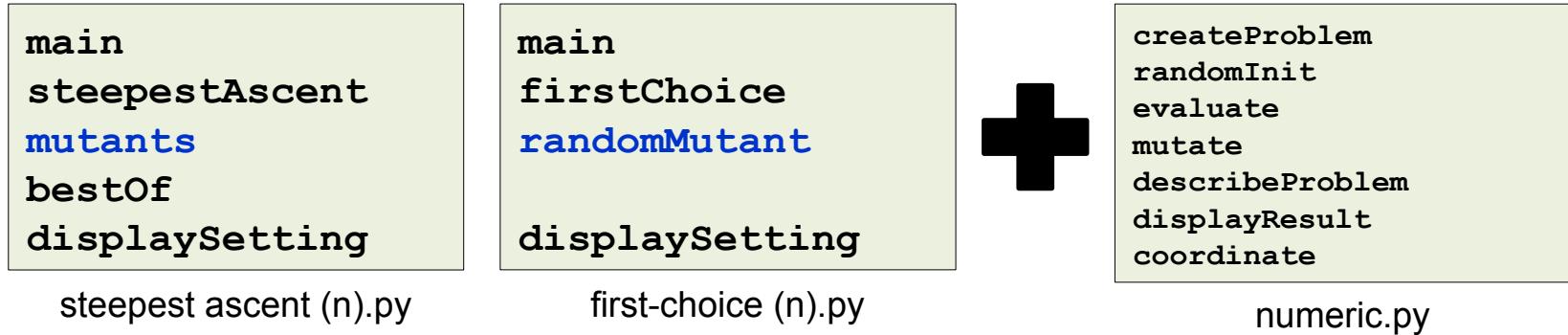
# **SEARCH TOOL V1**

# Modularity

- Numeric Optimization Problem
  - **Separate the common module** from the *first-choice(n).py* and *steepest ascent(n).py* files to **create a *numeric.py* file** and configure the original two files to **import** from it.
- TSP Problem
  - **Separate the common module** from the *first-choice(n).py* and *steepest ascent(n).py* files to **create a *tsp.py* file** and configure the original two files to **import** from it.

# Introducing ‘numeric’ Module (for Numerical Optimization)

- By moving duplicated codes of SAHC-N and FCHC-N to a separate module named ‘numeric’, we can easily reuse them in both programs by simply importing the module (공통으로 사용하는 로직을 분리하여 따로 저장 )
- Only a few functions remain in the main programs of SAHC-N and FCHC-N after the code migration (기존 코드가 간결해짐)      공통 (import)



- displaySetting in SAHC-N and that in FCHC-N are of the same purposes, but with slightly different print messages

Therefore... not separated  
into a common module

# TSP Problem

- The problem of planning the **route (= order of visits)** to visit all cities with the **shortest total distance**.
- Assume the problem is input via a **text file**, as shown below.
  - The **number of cities** is on the first line.
  - The subsequent lines contain the **coordinates of each city**.

| tsp30.txt |           |
|-----------|-----------|
| 1         | 30        |
| 2         | (8, 31)   |
| 3         | (54, 97)  |
| 4         | (50, 50)  |
| 5         | (65, 16)  |
| 6         | (70, 47)  |
| 7         | (25, 100) |
| 8         | (55, 74)  |
| 9         | (77, 87)  |
| 10        | (6, 46)   |

## Steepest-Ascent Hill Climbing for TSP

- Since the algorithm is the same as that for numerical optimization, the main program of SAHC-N may be reused for TSP without much change
  - main, steepestAscent, and bestof can be reused without any change at all
  - mutants should be implemented differently because the representation of candidate solution for TSP is different from that for numerical optimization
  - displaySetting should also be changed because there is no notion of mutation step size in solving TSPs
- We notice that we need a module like ‘numeric’ to be imported, but the codes in it should be changed appropriately for TSPs

## Steepest-Ascent Hill Climbing for TSP

- A new module named ‘**tsp**’ is created for being used as a replacement of the ‘**numeric**’ module
  - The functions **createProblem**, **randomInit**, **evaluate**, **describeProblem**, and **displayResult** in ‘**numeric**’ are also needed in ‘**tsp**’ but with different implementations
  - And there may be some new functions needed for solving TSPs
- **createProblem()**:
  - Gets a file name from the user and reads problem information from the file
    - Number of cities – saved as an integer
    - City locations – saved as a list of 2-tuples
  - Creates a matrix of distances between every pair of cities (**calcDistanceTable** – a new function)
  - Returns the triple: number of cities, locations, distance table

# Steepest-Ascent Hill Climbing for TSP

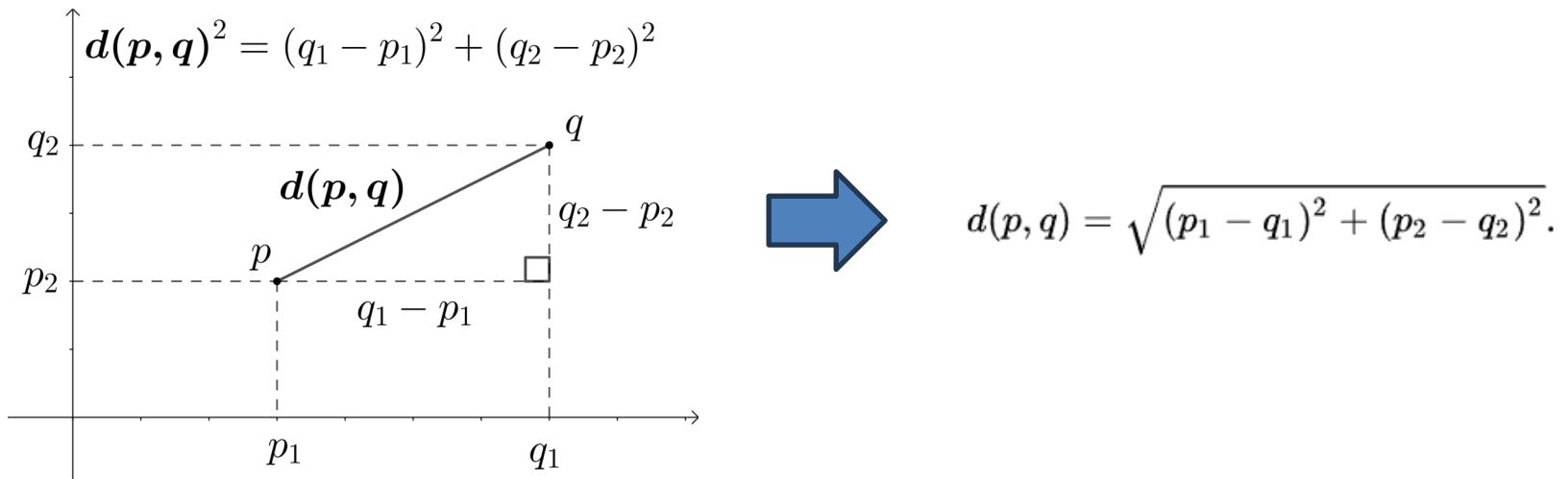
- **createProblem():**

```
def createProblem():
    ## Read in a TSP (# of cities, locations) from a file.
    ## Then, create a problem instance and return it.
    fileName = input("Enter the file name of a TSP: ")
    infile = open(fileName, 'r')
    # First line is number of cities
    numCities = int(infile.readline())
    locations = []
    line = infile.readline() # The rest of the lines are locations
    while line != '':
        locations.append(eval(line)) # Make a tuple and append
        line = infile.readline()
    infile.close()
    table = calcDistanceTable(numCities, locations)
    return numCities, locations, table
```

Read number of cities on first line as Integer

## Steepest-Ascent Hill Climbing for TSP

- **calcDistanceTable(numCities, locations):**
  - Calculates an  $n \times n$  matrix of pairwise distances based on **locations** ( $n = \text{numCities}$ ) // use Euclidean distance calculation



## Steepest-Ascent Hill Climbing for TSP

- **randomInit(p):**
  - Returns a randomly shuffled list of IDs of the cities in **p** (= random initial solution)

```
def randomInit(p):    # Return a random initial tour
    n = p[0]
    init = list(range(n))
    random.shuffle(init)
    return init
```

The shuffle function ensure the random list is a valid solution.

- **Evaluate(current, p):**
  - Calculates the tour cost (= 이동 거리 총합 ) of **current** by looking at the **distance matrix** given in **p**
  - Distance matrix for evaluation is the **table** output of **calcDistanceTable** function.

## Steepest-Ascent Hill Climbing for TSP

- **inversion(current, i, j):**
  - Makes a copy of `current`, inverts its subsection from  $i$  to  $j$ , and returns the mutant // 인덱스  $i$ 부터  $j$ 까지의 모든 item을 뒤집기
  - This function takes the role of `mutate` for numerical optimization // numerical optimization에서 mutate 함수의 역할을 수행

```
def inversion(current, i, j): # Perform inversion
    curCopy = current[:]
    while i < j:
        curCopy[i], curCopy[j] = curCopy[j], curCopy[i]
        i += 1
        j -= 1
    return curCopy
```

In Python, the exchange of two variables can be easily implemented.

From prev. lecture

# Ref) Variable swapping

- The general method for swapping two variables (using a temporary variable)

```
temp = var1;  
var1 = var2;  
var2 = temp;
```

- Method for swapping without a temporary variable:
  - (Arithmetic)

```
int x = 10, y = 5;  
  
// Code to swap 'x' and 'y'  
x = x + y; // x now becomes 15  
y = x - y; // y becomes 10  
x = x - y; // x becomes 5
```

## 2. XOR

```
int x = 10, y = 5;  
// Code to swap 'x' (1010) and 'y' (0101)  
x = x ^ y; // x now becomes 15 (1111)  
y = x ^ y; // y becomes 10 (1010)  
x = x ^ y; // x becomes 5 (0101)
```

## Steepest-Ascent Hill Climbing for TSP

- **describeProblem(p):**
  - Prints the number of cities in **p**, followed by the city locations, five locations per line

```
def describeProblem(p):  
    print()  
    n = p[0]  
    print("Number of cities:", n)  
    print("City locations:")  
    locations = p[1]  
    for i in range(n):  
        print("{0:>12}".format(str(locations[i])), end = ' ')  
        if i % 5 == 4:  
            print()
```

## Steepest-Ascent Hill Climbing for TSP

- **displayResult(solution, minimum):**
  - Displays **solution** (the best tour found) (**tenPerRow**), **minimum** (its evaluation value), and the total number of evaluations

```
def displayResult(solution, minimum):  
    print()  
    print("Best order of visits:")  
    tenPerRow(solution)      # Print 10 cities per row  
    print("Minimum tour cost: {:.0f}".format(round(minimum)))  
    print()  
    print("Total number of evaluations: {:.0f}".format(NumEval))
```

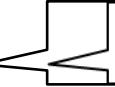
- **tenPerRow(solution):**
  - Prints city ids in order of visit, ten ids per row
  - 최종 결과 (= 도시 방문 순서 ) 를 10 개씩 화면에 출력하는 함수

## Steepest-Ascent Hill Climbing for TSP

- **tenPerRow(solution):**
  - Prints city ids in order of visit, ten ids per row
  - 최종 결과 (= 도시 방문 순서 ) 를 한 줄에 10 개씩 화면 출력

```
def tenPerRow(solution):  
    for i in range(len(solution)):  
        print("{0:>5}".format(solution[i]), end=' ')  
        if i % 10 == 9:  
            print()
```

## Steepest-Ascent Hill Climbing for TSP

- Below, we describe how mutants in the main program is implemented differently for TSPs than numerical optimization
  - 현재의 솔루션을 기준으로, 이웃하는 솔루션을 생성하는 방법 설명
- **mutants (current, p)**:  Reference: In the numerical optimization problem, the mutant function generates  $2n$  neighbors.
- Returns  $n$  neighbors of **current** ( $n = \text{number of cities in } p$ ),
  - Each neighbor is generated by inverting the subsection beginning from  $i$  and ending at  $j$  (**inversion**), where the  $(i, j)$ -pair is chosen randomly
  - The inversion for  $(i, j)$ -pair is applied only when  $i \neq j$  and the pair has never been tried before

## Steepest-Ascent Hill Climbing for TSP

- **mutants(current, p):**

```
def mutants(current, p): # Apply inversion
    n = p[0]
    neighbors = []
    count = 0
    triedPairs = []
    while count <= n: # Pick two random loci for inversion
        i, j = sorted([random.randrange(n) for _ in range(2)])
        if i < j and [i, j] not in triedPairs:
            triedPairs.append([i, j])
            curCopy = inversion(current, i, j)
            count += 1
            neighbors.append(curCopy)
    return neighbors
```

## First-Choice Hill Climbing for TSP

- The main program of FCHC-N can be reused without much change
  - `main` and `firstChoice` can be reused without change
  - `displaySetting` needs to be changed because the mutation step size is now irrelevant
  - `randomMutant` should be implemented differently because of the different representation of candidate solution for TSPs
- `randomMutant(current, p)`:
  - Returns a mutant of `current`, which is made by inverting the sub-section beginning from `i` and ending at `j` (`inversion`), where `i` and `j` are chosen randomly

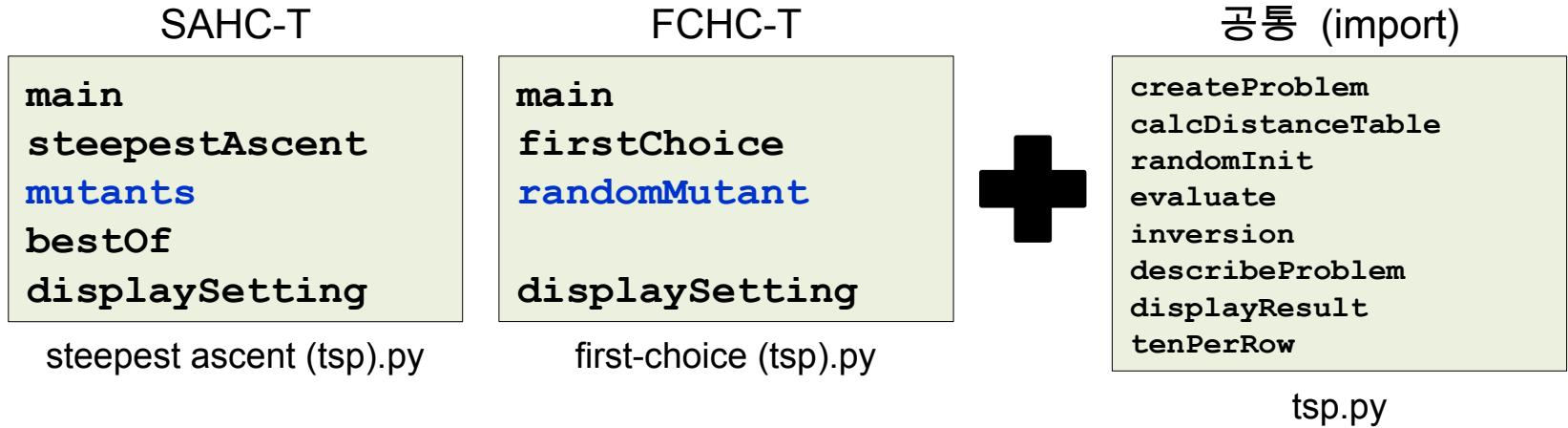
## First-Choice Hill Climbing for TSP

- **randomMutant(current, p):**
  - Returns a mutant of `current`, which is made by inverting the subsection beginning from `i` and ending at `j` (`inversion`), where `i` and `j` are chosen randomly

```
def randomMutant(current, p): # Apply inversion
    while True:
        i, j = sorted([random.randrange(p[0])
                      for _ in range(2)])
        if i < j:
            curCopy = inversion(current, i, j)
            break
    return curCopy
```

# Search Tool v1 for TSP

- Code configuration after modularization



# TSP Problem

- Execution Results (Example using the **First-Choice Algorithm**)

```
Enter the file name of a TSP: problem/tsp30.txt
```

Input the name and path of the .txt file defining the problem (User Input).

Number of cities: 30

City locations:

|           |           |          |          |          |
|-----------|-----------|----------|----------|----------|
| (8, 31)   | (54, 97)  | (50, 50) | (65, 16) | (70, 44) |
| (25, 100) | (55, 74)  | (77, 87) | (6, 46)  | (70, 78) |
| (13, 38)  | (100, 32) | (26, 35) | (55, 16) | (26, 77) |
| (17, 67)  | (40, 36)  | (38, 27) | (33, 2)  | (48, 9)  |
| (62, 20)  | (17, 92)  | (30, 2)  | (80, 75) | (32, 26) |
| (43, 79)  | (57, 49)  | (18, 24) | (96, 76) | (81, 1)  |

City 4 (starts from index 0)

The last city is City 29

Search algorithm: **First-Choice** Hill Climbing

Max evaluations with no improvement: 100 iterations

Best order of visits:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 4  | 26 | 2  | 24 | 16 | 27 | 0  | 8  | 10 | 12 |
| 17 | 18 | 22 | 19 | 3  | 13 | 20 | 29 | 11 | 28 |
| 23 | 7  | 9  | 6  | 25 | 1  | 14 | 15 | 21 | 5  |

The Optimal Order of Visits

Minimum tour cost: 491

Total number of evaluations: 677

The Total Distance traveled when following the optimal route.

# TSP Problem

- Execution Results (Example using the **Steepest-Ascent Algorithm**)

Enter the file name of a TSP: problem/tsp30.txt

Input the name and path of the .txt file defining the problem (User Input).

Number of cities: 30

City locations:

|           |           |          |          |          |
|-----------|-----------|----------|----------|----------|
| (8, 31)   | (54, 97)  | (50, 50) | (65, 16) | (70, 47) |
| (25, 100) | (55, 74)  | (77, 87) | (6, 46)  | (70, 78) |
| (13, 38)  | (100, 32) | (26, 35) | (55, 16) | (26, 77) |
| (17, 67)  | (40, 36)  | (38, 27) | (33, 2)  | (48, 9)  |
| (62, 20)  | (17, 92)  | (30, 2)  | (80, 75) | (32, 36) |
| (43, 79)  | (57, 49)  | (18, 24) | (96, 76) | (81, 39) |

Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 11 | 29 | 20 | 13 | 3  | 4  | 28 | 23 | 9  | 1  |
| 7  | 6  | 2  | 16 | 26 | 25 | 21 | 5  | 8  | 12 |
| 14 | 10 | 0  | 12 | 27 | 24 | 17 | 19 | 18 | 22 |

The Optimal Order of Visits

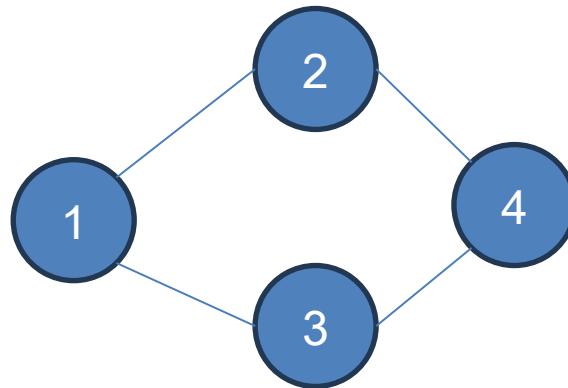
Minimum tour cost: 620

Total number of evaluations: 807

The **Total Distance** traveled when following the optimal route.

# Ref) Adjacency Matrix / 인접 행렬

- Adjacency Matrix
  - One of the effective ways to represent an **Undirected Graph**.
  - Directly adjacent vertices  $(i, j)$  are represented by **1**, and two vertices that are not adjacent are represented by **0**.
  - The entry for  $(i, i)$  (a vertex's adjacency to itself) is represented by **0**.



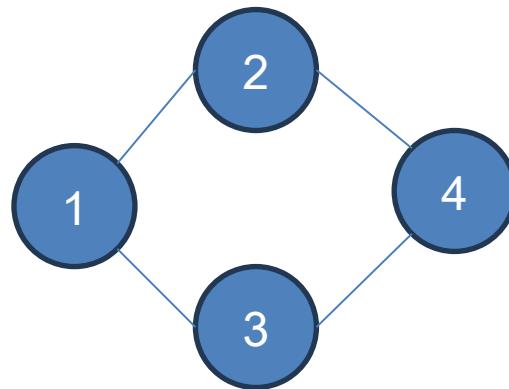
|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |

# Ref) Adjacency Matrix / 인접 행렬

- Formal Definition
  - In graph theory and computer science, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.
  - In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected (i.e. all of its edges are bidirectional), the adjacency matrix is symmetric.
    - simple graph : a graph that does not have more than one edge between any two vertices and no edge starts and ends at the same vertex

# Ref) Adjacency Matrix / 인접 행렬

- Interesting properties of Adjacency Matrix : Matrix powers
  - Let  $A$  be the Adjacency Matrix representing the graph below



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

- In this case, the result of  $A^2$  is shown on the left, and the result of  $A^3$  is shown on the right. **What is the meaning of each result??**

|   |   |   |   |
|---|---|---|---|
| 2 | 0 | 0 | 2 |
| 0 | 2 | 2 | 0 |
| 0 | 2 | 2 | 0 |
| 2 | 0 | 0 | 2 |

|   |   |   |   |
|---|---|---|---|
| 0 | 4 | 4 | 0 |
| 4 | 0 | 0 | 4 |
| 4 | 0 | 0 | 4 |
| 0 | 4 | 4 | 0 |

# Ref) Adjacency Matrix / 인접 행렬

- Interesting properties of Adjacency Matrix : Matrix powers

Unless specified, all graphs are assumed to be simple and connected, that is, there is at most one edge between each pair of vertices, there are no loops, and there is at least one path between every two vertices. The adjacency matrix  $A$  or  $A(G)$  of a graph  $G$  having vertex set  $V = V(G) = \{1, \dots, n\}$  is an  $n \times n$  symmetric matrix  $a_{ij}$  such that  $a_{ij} = 1$  if vertices  $i$  and  $j$  are adjacent and 0 otherwise.

## Powers of the Adjacency Matrix

The following well-known result will be used frequently throughout:

**Theorem 0.1** *The  $(i, j)^{th}$  entry  $a_{ij}^k$  of  $A^k$ , where  $A = A(G)$ , the adjacency matrix of  $G$ , counts the number of walks of length  $k$  having start and end vertices  $i$  and  $j$  respectively.*

$A^k$  계산 결과에서  $(i, j)$  번째 값이  $n$  이라면,  $i$ 에서 출발해서  
 $i$ 까지 도착하는 서로 다른 경로의 수가  $n$ 개라는 의미

끝 .



오늘 강의는 여기까지 !

# Search Algorithms: Object-Oriented Implementation (Part C)

# Contents

- Conventional vs. AI Algorithms
- Local Search Algorithms
- Implementing Hill-Climbing Algorithms
- Defining ‘Problem’ Class // today’ topic (Part C-1)
- Adding Gradient Descent // next week (Part C-2)
- Defining ‘HillClimbing’ Class
- Adding More Algorithms and Classes
- Adding Genetic Algorithm
- Experiments

Class implementation

# MIGRATING INTO CLASSES

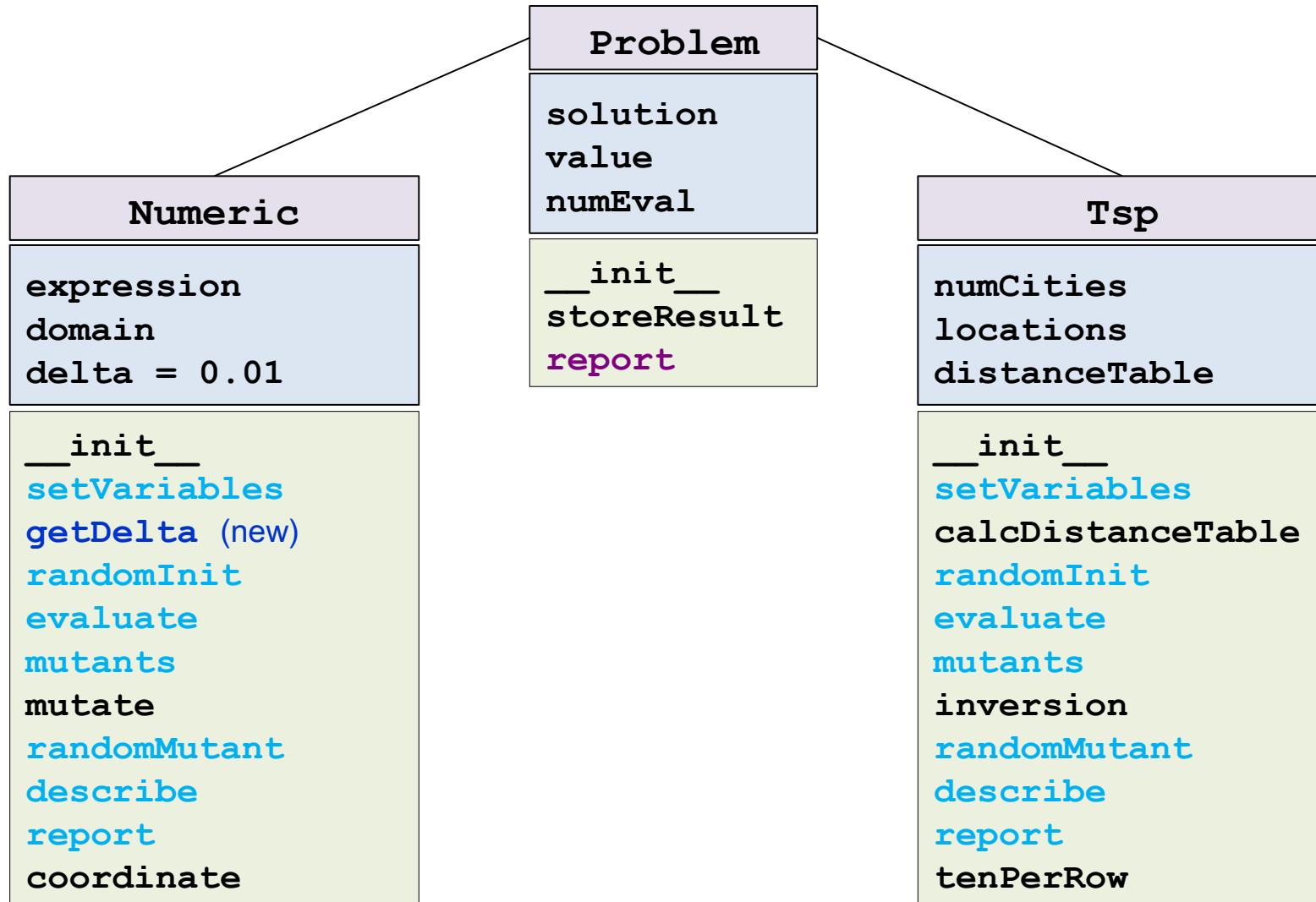
# Implementing Class

- Basic Idea
  - Convert the content saved in the numeric.py and tsp.py code into Numeric and Tsp classes, respectively.
  - If we convert the existing code as is into classes, Numeric Class and Tsp Class will contain functions that used for the following: 1) The problem to be solved, 2) The optimal solution, 3) Algorithm for finding the solution.
  - At this point, Numeric and Tsp Class commonly include variables for storing the optimal solution and functions for printing the solution to the screen.
    - **Solution using Inheritance:** Even if new types of problems and new kinds of solution algorithms are added in the future, the logic for **storing the solution and printing the solution to the screen** will be **commonly used**. Therefore, this common logic is separated to **create a Problem class** for managing the solution storage variables and output logic.
    - Then, **generate subclasses** (or child classes) like **Numeric** and **Tsp** by **inheriting** from the **Problem class**.

# Implementing Class

- Implementation approach
  - Top-Level Class : Problem (`problem.py`)
    - Implements variables and functions related to the solution.
  - Subclasses : Numeric, Tsp (`same file alongside problem.py`)
    - Inherit from the **Problem class** (inheriting solution-related variables/functions).
    - Implement variables for **storing the Numeric/TSP problem itself**.
    - Implement functions that are **frequently/commonly** used by solution algorithms for solving the Numeric/TSP problem.
  - Solution Algorithm Code: `first-choice(n).py`, `first-choice(tsp).py`, `steepest ascent(n).py`, `steepest ascent(tsp).py`
    - Includes the main function of the existing code.
    - Includes logic specialized for each solution algorithm.

# Defining Classes



# Defining Classes

- However, the actual class implementation will include more methods than the diagram on the previous page.
  - Problem Class
    - Functions such as `setVariables`, `randomInit`, `evaluate`, `mutants`, `randomMutant`, and `describe` are methods that the Numeric and Tsp Subclasses must implement and use. Therefore, these methods are declared in the Problem class and left as placeholders (`pass`)
    - Subclasses (Numeric, Tsp) will actually implement the methods with **method overriding**

```
def randomMutant(self, current):
    pass

def describe(self):
    pass

def storeResult(self, solution, value):
    self._solution = solution
    self._value = value

def report(self):
    print()
    print("Total number of evaluations: {}".format(
```

```
class Problem:
    def __init__(self):
        self._solution = []
        self._value = 0
        self._numEval = 0

    def setVariables(self):
        pass

    def randomInit(self):
        pass

    def evaluate(self):
        pass

    def mutants(self):
        pass
```

# Defining Classes

- However, the actual class implementation will include more methods than the diagram on the previous page.
  - Numeric Class
    - Since the Gradient Descent Algorithm will be added in the future, related methods are: `gradient`, `takeStep`, `isLegal`
  - Tsp Class
    - Almost identical to the existing `tsp.py`
  - Some of the previously used methods names were changed:
    - `createProblem` → `setVariables`
    - `describeProblem` → `describe`
    - `displayResult` → `report`

# Defining Classes

- Four source codes are used to implement and run the solution algorithms:
  - `first-choice (n).py`
  - `steepest ascent (n).py`
  - `first-choice (tsp).py`
  - `steepest ascent (tsp).py`
- In addition, the following source code is added for the implementation of the Gradient Descent (GD) solution algorithm:
  - `gradient descent.py`
  - GD can only be applied to **continuous variables**, so it can only be used for **Numeric Optimization** problems and **cannot be used for TSP** problems.

# Defining Classes

- The subclass `Numeric` has three variables for storing specifics about the problem:
  - `expression`: function expression of a numerical optimization problem
  - `domain`: lower and upper bounds of each variable of the function
  - `delta`: step size of axis-parallel mutation

```
class Numeric(Problem):  
    def __init__(self):  
        Problem.__init__(self)  
        self._expression = ''  
        self._domain = []      # domain as a list  
        self._delta = 0.01     # Step size for axis-parallel mutation
```

# Defining Classes

- The subclass **Numeric** has three variables for storing specifics about the problem:

```
class Numeric(Problem):  
    def __init__(self):  
        Problem.__init__(self)  
        self._expression = ''  
        self._domain = []      # domain as a list  
        self._delta = 0.01     # Step size for axis-parallel mutation
```

- delta** takes the role of **DELTA** that was previously the named constant of the ‘numeric’ module
  - delta** is referred to by **mutants** and **randomMutant**
  - Its value is preset to 0.01 for the time being for convenience
  - (getter method) The method **getDelta** is newly made for being used by the **displaySetting** function of the main program when displaying the value of **delta**

```
def getDelta(self):  
    return self._delta
```

# Defining Classes

- The subclass `Tsp` also has three variables:
  - `numCities`: number of cities (N)
  - `locations`: coordinates of city locations in a  $100 \times 100$  square
  - `distanceTable`: matrix of distances of every pair of cities (N-by-N)

```
class Tsp(Problem):
    def __init__(self):
        Problem.__init__(self)
        self._numCities = 0
        self._locations = []          # A list of tuples
        self._distanceTable = []
```

# Defining Classes

- Both `Numeric` and `Tsp` have the `setVariables` method that reads in the specifics of the problem to be solved and stores them in the relevant class variables
  - It is renamed from its previous version `createProblem`
  - Unlike `createProblem`, `setVariables` does not return anything
    - Because problem-relation information is stored in the class member variables.

# Defining Classes

- Notice that the functions `mutants` and `randomMutant` can also be converted to methods of both `Numeric` and `Tsp` classes
  - `mutants`, `randomMutant` are declared in the `Problem` Class (as placeholders), but the actual implementation is performed in the `Numeric` and `tsp` classes.
    - `mutants` appears in both `SAHC-N` and `SAHC-T`
    - `randomMutant` appears in both `FCHC-N` and `FCHC-T`

# Defining Classes

- The base class **Problem** has three variables for storing the result of search:
  - solution**: final solution found by the search algorithm,  $x^*$
  - value**: its objective value,  $f(x^*)$
  - numEval**: total number of evaluations taken for the search
- The values of **solution** and **value** are set by the **storeResult** method that is called by the search algorithms
  - Now the search algorithms do not have the **return** statement
- But, the way to report the optimal solution differs depending on the problem type.
  - The common solution report of **Problem.report** are shared between **Numeric.report** and **Tsp.report** with slight changes for respective problem.

```
class Problem:  
    def __init__(self):  
        self._solution = []  
        self._value = 0  
        self._numEval = 0
```

```
def report(self): # Numeric.report  
    print()  
    print("Solution found:")  
    print(self.coordinate()) # Convert list to tuple  
    print("Minimum value: {:.3f}".format(self._value))  
    Problem.report(self)
```

```
def report(self): # Tsp.report  
    print()  
    print("Best order of visits:")  
    self.tenPerRow() # Print 10 cities per row  
    print("Minimum tour cost: {:.3f}".format(round(self._value)))  
    Problem.report(self)
```

```
def report(self): # Problem.report  
    print()  
    print("Total number of evaluations:
```

# Defining Classes

- **report** (previously `displayResult`) is defined in both the base class and the subclasses
  - **report** in the base class prints `numEval` that is a common information to both the subclasses
    - The one in the base case handles general information and is inherited to the subclasses
  - **report** in each subclass prints further information specific to the problem type
  - To call a method of a superclass, it should be stated explicitly (e.g., `Problem.report(self)`)
- We can see that object-oriented programming provides us with the opportunity to organize codes in a way easier to maintain

# Defining Classes

- The value of `numEval` is initially 0 and incremented whenever the `evaluate` method is called
  - Previously, reporting the value of the global variable `NumEval` was done by `displayResult` of both ‘numeric’ and ‘tsp’ modules
  - But, `NumEval` appeared in duplicate in both modules and thus, printing `numEval` is done by the `Problem.report` method now

```
def report(self): # Problem.report
    print()
    print("Total number of evaluations: {}".format(self._numEval))
```

# Defining Classes

- `Numeric` and `Tsp` have many **methods of the same names** but with different implementations (declared in Problem, implemented in Numeric and TSP)
  - **Polymorphism** allows us to write codes that look the same regardless of the type of problem to be solved
  - E.g., `evaluate` of `Numeric` and `evaluate` of `TSP` are of the same name but are implemented differently depending on the type of problem to be solved (numerical optimization or TSP)
- By introducing the classes and taking advantage of polymorphism, we will eventually be able to unite the main programs of different search algorithms into a single program
  - Duplications among different main programs can be avoided

# Defining Classes

- We store **Problem** class in a separate file named ‘problem.py’
  - ‘random.py’ and ‘math.py’ that were imported to the previous modules should now be imported to the ‘problem.py’ file
  - Each main program needs to import either **Numeric** or **Tsp** from the ‘problem.py’ file
    - ex) `from problem import Numeric`
    - ex) `from problem import Tsp`

# Changes to the Main Program

- To execute different solution algorithms for each problem type, we use the corresponding Python code files:
  - first-choice (n).py
  - first-choice (tsp).py
  - steepest ascent (n).py
  - steepest ascent (tsp).py
  - (later on, gradient descent.py)
- Each code file is simplified and contains only the following content:
  - main function // Runs the algorithm from the file, executes the algorithm and display result.
  - Core algorithm implementation// Logics specialized for that algorithm.
  - Function that prints the setting values// Setting values for that algorithm.

# Code outside of problem.py

**steepest ascent(tsp).py**

```
def main():
    p = Tsp()
    ...
def steepestAscent(p):
    ...
def bestOf(neighbors,p):
    ...
def displaySetting():
    ...

main()
```

```
def main():
    # Create an object for TSP
    p = Tsp()          # Create a problem
    p.setVariables()  # Set its class var
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algorithm set
    p.describe()
    displaySetting()
    # Report results
    p.report()
```

**first-choice(tsp).py**

```
def main():
    p = Tsp()
    ...
def firstChoice(p):
    ...
def displaySetting():
    ...

main()
```

```
def main():
    # Create an object for TSP
    p = Tsp()          # Create a problem
    p.setVariables()  # Set its class var
    # Call the search algorithm
    firstChoice(p)
    # Show the problem to be solved
    p.describe()
    displaySetting()
    # Report results
    p.report()
```

# Code outside of problem.py

**steepest ascent(n) .py**

```
def main():
    p = Numeric()
    ...
def steepestAscent(p):
    ...
def bestOf(neighbors,p):
    ...
def displaySetting(p):
    ...

main()
```

**first-choice(n) .py**

```
def main():
    p = Numeric()
    ...
def firstChoice(p):
    ...
def displaySetting(p):
    ...

main()
```

```
def main():
    # Create a Problem object for
    p = Numeric()      # Create a p
    p.setVariables()  # Set its cl
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algori
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

```
def main():
    # Create a Problem object for
    p = Numeric()      # Create a p
    p.setVariables()  # Set its cl
    # Call the search algorithm
    firstChoice(p)
    # Show the problem and algori
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

# Code outside of problem.py

gradient\_descent.py

```
def main():
    p = Numeric()
    ...
def gradientDescent(p):
    ...
def displaySetting(p):
    ...

main()
```

Next lecture  
topic

```
def main():
    # Create a Problem object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables()   # Set its class variables (expression, domain)
    # Call the search algorithm
    gradientDescent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

# Changes to the Main Program

- It is the `main` function that creates an instance of `Problem` object (`p = Numeric()` or `p = Tsp()`)
- Then, the `setVariables` method (`p.setVariables`) is executed to store the problem-related values in the relevant class variables
  - Stores problem-related information in the **class member variables**.
  - Previously this was done by the function `createProblem` (function name changed)

```
def main(): ex) Steepest Ascent for Numeric Optimization
    # Create a Problem object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables()  # Set its class variables (expression, domain)
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

# Changes to the Main Program

- After creating problem `p`, the `main` function calls the search algorithm with `p` as its argument
  - The search algorithm calls the methods such as `randomInit`, `evaluate`, and `mutants` to conduct the search and these methods refer to the relevant class variables when executed

```
def main():    ex) Steepest Ascent for Numeric Optimization
    # Create a Problem object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables()  # Set its class variables (expression, domain)
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algorithm
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

The source code implementing each algorithm is an independent function (e.g., `steepestAscent`) that receives the class instance `p` as an argument.

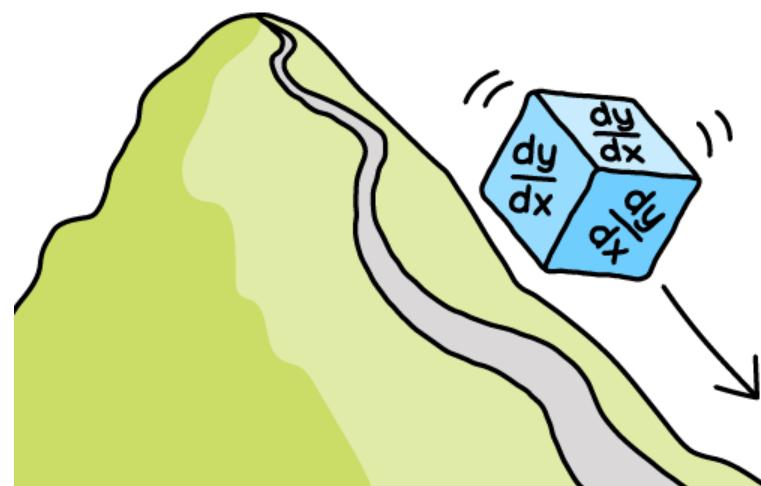
# Changes to the Main Program

- After running the search algorithm, the `main` function makes calls to relevant methods
  - to show the specifics of the problem solved (`p.describe`),
  - to display the settings of the search algorithm (`displaySetting`),
  - and then to display the result (`p.report`)

```
def main(): ex) Steepest Ascent for Numeric Optimization
    # Create a Problem object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables()  # Set its class variables (expression, domain)
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

## Changes to the Main Program

- There were two versions of `mutants` for the steepest-ascent hill climbing: one for numeric optimization and the other for TSPs
  - They are now migrated to the `Numeric` and `Tsp` classes
- Similarly, `randomMutant` of the first-choice hill climbing are migrated to the `Numeric` and `Tsp` classes, too
- The functions `displaySetting` and `bestOf` still remain in the main program because they are tied with the search algorithms used rather than the types of the problems solved



Solution for solving continuous variable problems

# GRADIENT DESCENT ALGO.

# Search Algorithms: Object-Oriented Implementation (Part C)

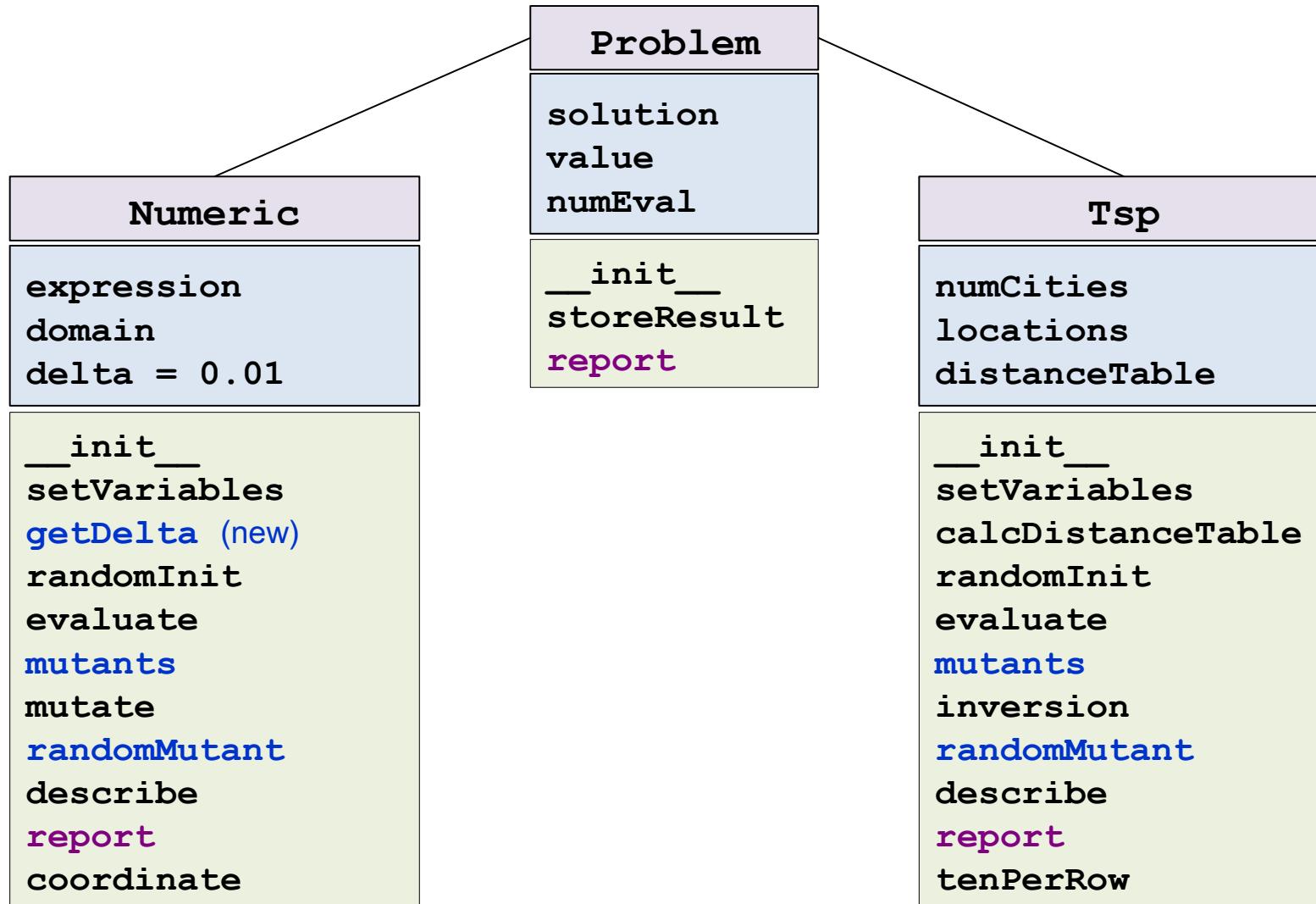
# Contents

- Conventional vs. AI Algorithms
- Local Search Algorithms
- ~~Genetic Algorithm~~
- Implementing Hill-Climbing Algorithms
- Defining ‘Problem’ Class // 지난 강의 (Part C-1)
- Adding Gradient Descent // 오늘 강의 (Part C-2)
- Defining ‘HillClimbing’ Class
- Adding More Algorithms and Classes
- Adding Genetic Algorithm
- Experiments

Class implementation

# MIGRATING TO CLASSES

# Defining Classes



# Code outside of problem.py

**steepest ascent(tsp).py**

```
def main():
    p = Tsp()
    ...
def steepestAscent(p):
    ...
def bestOf(neighbors,p):
    ...
def displaySetting():
    ...

main()
```

```
def main():
    # Create an object for TSP
    p = Tsp()          # Create a problem
    p.setVariables()  # Set its class var
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algorithm set
    p.describe()
    displaySetting()
    # Report results
    p.report()
```

**first-choice(tsp).py**

```
def main():
    p = Tsp()
    ...
def firstChoice(p):
    ...
def displaySetting():
    ...

main()
```

```
def main():
    # Create an object for TSP
    p = Tsp()          # Create a problem
    p.setVariables()  # Set its class var
    # Call the search algorithm
    firstChoice(p)
    # Show the problem to be solved
    p.describe()
    displaySetting()
    # Report results
    p.report()
```

# Code outside of problem.py

**steepest ascent(n) .py**

```
def main():
    p = Numeric()
    ...
def steepestAscent(p):
    ...
def bestOf(neighbors,p):
    ...
def displaySetting(p):
    ...

main()
```

**first-choice(n) .py**

```
def main():
    p = Numeric()
    ...
def firstChoice(p):
    ...
def displaySetting(p):
    ...

main()
```

```
def main():
    # Create a Problem object for
    p = Numeric()      # Create a p
    p.setVariables()  # Set its cl
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algori
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

```
def main():
    # Create a Problem object for
    p = Numeric()      # Create a p
    p.setVariables()  # Set its cl
    # Call the search algorithm
    firstChoice(p)
    # Show the problem and algori
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

# Code outside of problem.py

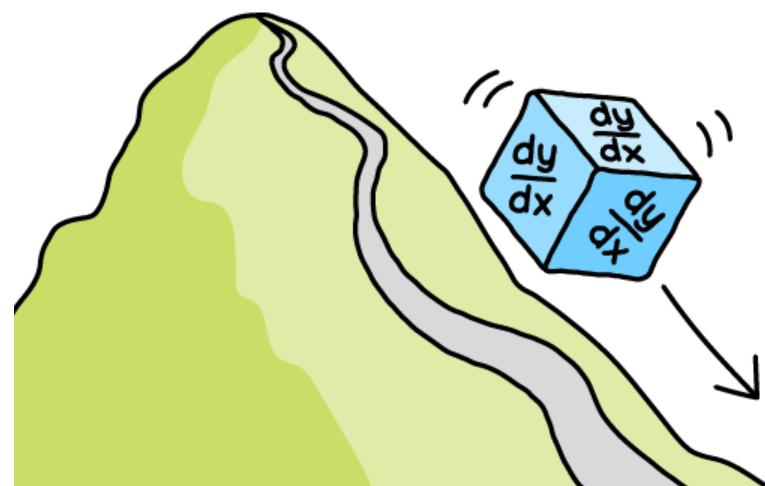
gradient\_descent.py

```
def main():
    p = Numeric()
    ...
def gradientDescent(p):
    ...
def displaySetting(p):
    ...

main()
```

Today's topic

```
def main():
    # Create a Problem object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables()   # Set its class variables (expression, domain)
    # Call the search algorithm
    gradientDescent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```



Solution for solving continuous variable problems

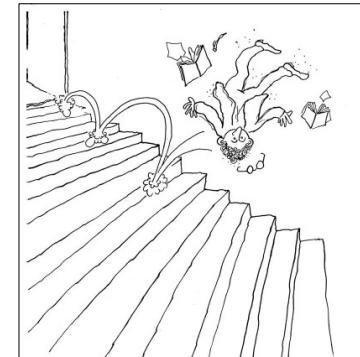
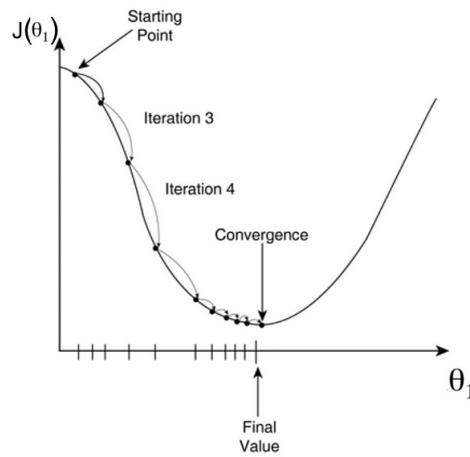
# GRADIENT DESCENT ALGO.

# Gradient Descent Algorithm

- General Descent Algorithm (When the objective is to **minimize**)
  - Finds the optimal **solution** by iteratively moving the variable's value slightly in the direction that makes the **objective function smaller**.
  - Find the direction **d** that reduces the objective function and change the variable's value by the **step size**.

```
x = random() # decision variable init
Repeat:
    x = x - α × d, α:step size (learning rate)
Until (Stop condition)
Return x
```

- Primarily used in **Optimization, Machine Learning, and Deep Learning**.
- Depending on how the **initial value** is set, the algorithm may converge to a **global optimum** (전역 해) or a **local optimum** (지역 해).



# Gradient Descent Algorithm

- **Gradient Descent** is referred to as a **first-order iterative algorithm** and is a representative technique for finding a **local optimum**.
  - Note: in case of convex optimization, local opt = global opt
  - Note: The search direction is determined using the **first derivative** (1 차 미분 = **gradient**).
  - [https://en.wikipedia.org/wiki/Gradient\\_descent](https://en.wikipedia.org/wiki/Gradient_descent)
  - This lecture assumes a **differentiable objective function**.

```
x = random() # decision variable init
Repeat:
    x = x - α × d, α:step size
Until (Stop condition)
Return x
```

The Gradient Descent algorithm uses the **gradient** (derivative) to find the direction **d** that reduces the objective function.

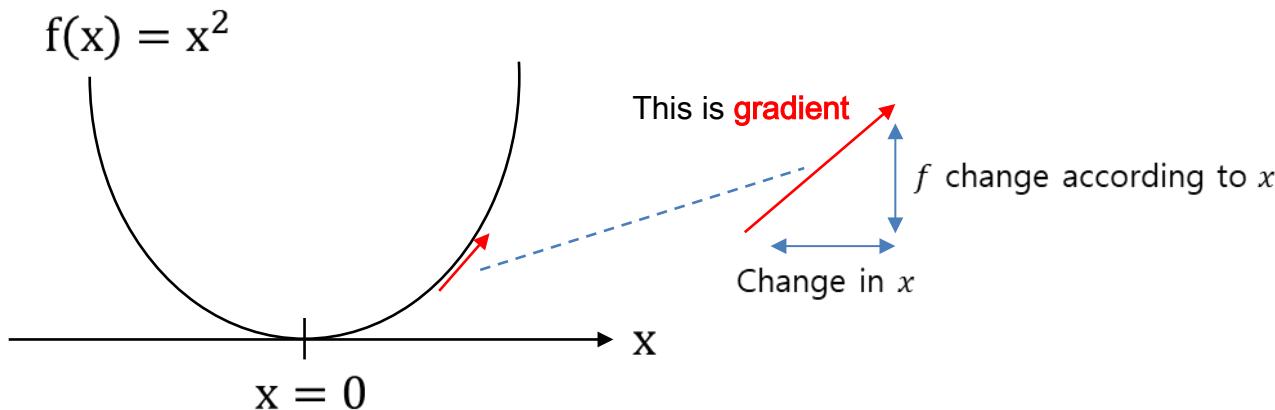
```
x = random() # decision variable init
Repeat:
    x = x - α × d, α:step size
Until (Stop condition)
Return x
```

The Gradient Descent algorithm uses the **gradient** (derivative) to find the direction **d** that reduces the objective function.

# Gradient Descent Algorithm

- The Direction to Decrease the Objective Function?
  - 1st derivative = **gradient** :

$$f'(x) = \lim_{\Delta \rightarrow 0} \frac{f(x + \Delta) - f(x)}{\Delta} = \frac{\text{change in } f}{\text{change in } x}$$



- The **Gradient** indicates the direction in which the function's value **increases**. Therefore, the **opposite direction of the gradient** ( $\text{grad } x - 1$ ) is the direction in which the **objective function decreases!**

# Gradient Descent Algorithm

- A general algorithm for finding solutions to optimization problems: Descent Algorithm
  - Searches for the solution by moving a certain **step size** in the **opposite direction of the Gradient.**

```
x = random() # decision variable init
Repeat:
    x = x + α × (−1 * ∇_x f₀(x)), α:step size, ∇:1st derivative
Until (Stop condition)
Return x
```

The "Stopping condition" can be set in various ways. It can be implemented to terminate the process if the result of plugging the newly calculated  $x$  into the objective function is not better than the previous result.

```
x = random() # decision variable init
Repeat:
    x = x + α × (−1 * ∇_x f₀(x)), α:step size, ∇:1st derivative
Until (Stop condition)
Return x
```

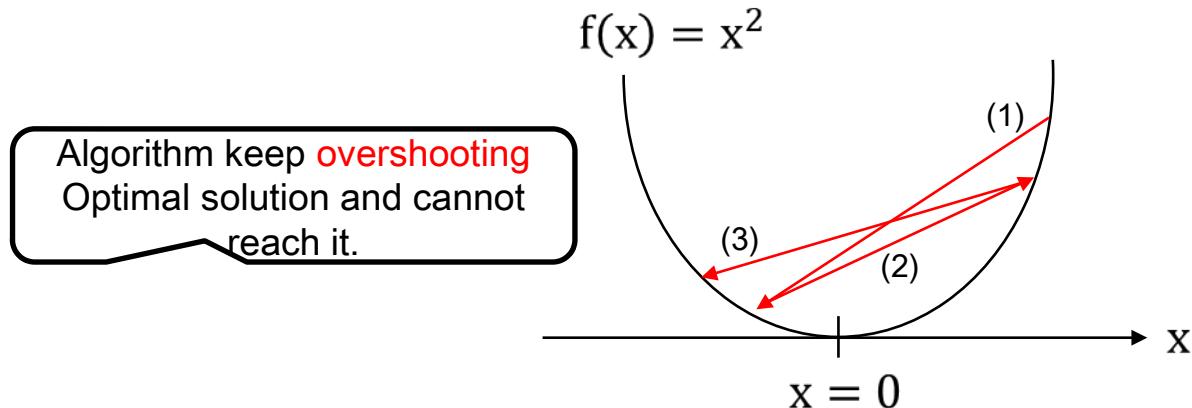
The "Stopping condition" can be set in various ways. It can be implemented to terminate the process if the result of plugging the newly calculated  $x$  into the objective function is not better than the previous result.

- The **step size** determines how much the decision variable will move in each iteration,  
In Machine and Deep Learning, this is called the **learning rate (lr)**.

```
keras.optimizers.Adam(
    learning_rate=0.001,
```

# Gradient Descent Algorithm

- If the step size(learning rate) is too small, it takes a long time to find the optimal solution
- If the step size(learning rate) is too large? **over-shooting** may occurs:



- So... how should the step size be?

# Gradient Descent Algorithm

- How should the **step size** be set? (here are the options...)
  - Select small value “appropriately” between (0,1): fixed value of around 0.01~0.001 are commonly used
  - Starting with a large value and gradually decreasing (ex: 1/iter\_count)
$$\alpha^t = \frac{1+m}{t+m} \text{ for } m \in \mathbb{R}_+$$
  - In general, stochastic gradient converges to a stationary point if
    - Ratio of sum of squared step-sizes over sum of step-sizes converges to 0 (\*1)
    - $\frac{\sum_{t=1}^{\infty} (\alpha_t)^2}{\sum_{t=1}^{\infty} \alpha_t} = 0$
  - Note: The method primarily used in **Deep Learning** is to dynamically update the step size (learning rate) using optimizers such as *AdaGrad*, *RMSProp*, and *Adam*

(\*1) <https://www.cs.ubc.ca/~fwood/CS340/lectures/L24.pdf>

# Gradient Descent Algorithm

- The Algorithm for Iteratively Updating the Variable's Value:

$$\mathbf{x}_{\text{new}} \leftarrow \mathbf{x}_{\text{current}} - \alpha \nabla f(\mathbf{x}_{\text{current}})$$

- While you can directly derive the **derivative** ( $\nabla f$ ) of the objective function, you can also calculate an **approximation** of the derivative value using a simple operation.

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

# Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Define the Objective Function and its Derivative Function

```
def f(x): # 목적 함수
    return 2 * (x**2)

def derivative(x): # 목적 함수의 미분
    return 4*x
```

# Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Implementing GD algorithm

```
import random

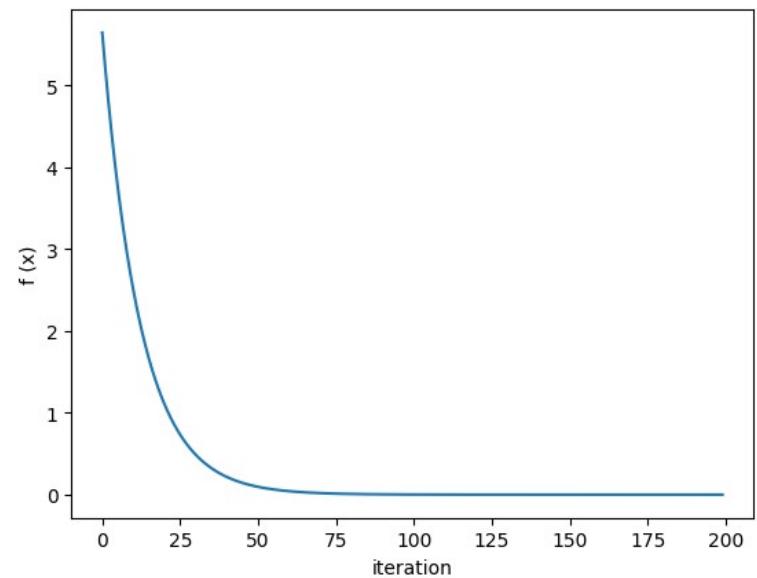
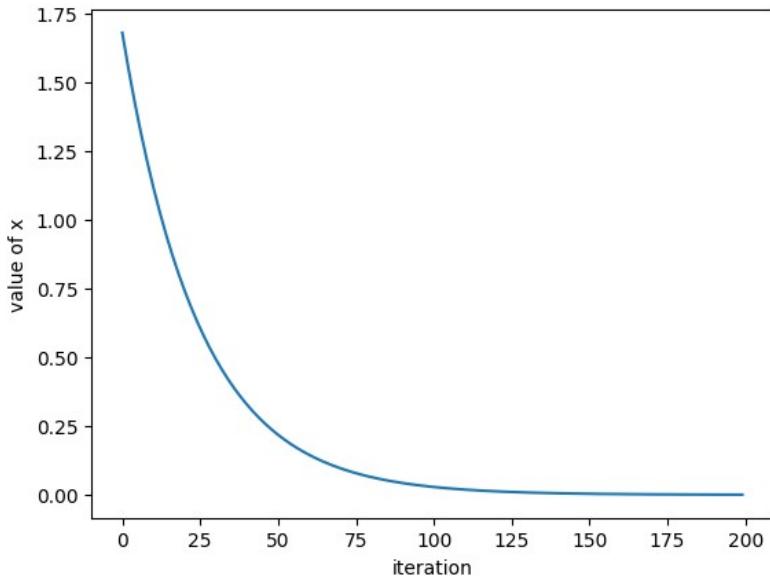
learning_rate = 0.01
#x_current = random.random()*4 - 2 # 무작위 시작점: [-2.0, +2.0)
x_current = 1.75 # 공평한 비교 실험을 위해, 시작점을 고정함

def GD(x_current):
    x_new = x_current - learning_rate * derivative(x_current)
    return x_new, f(x_new)

xs, fs, ITER_MAX = [], [], 200
for i in range(ITER_MAX):
    x_new, f_new = GD(x_current)
    xs.append(x_new)
    fs.append(f_new)
    x_current = x_new
```

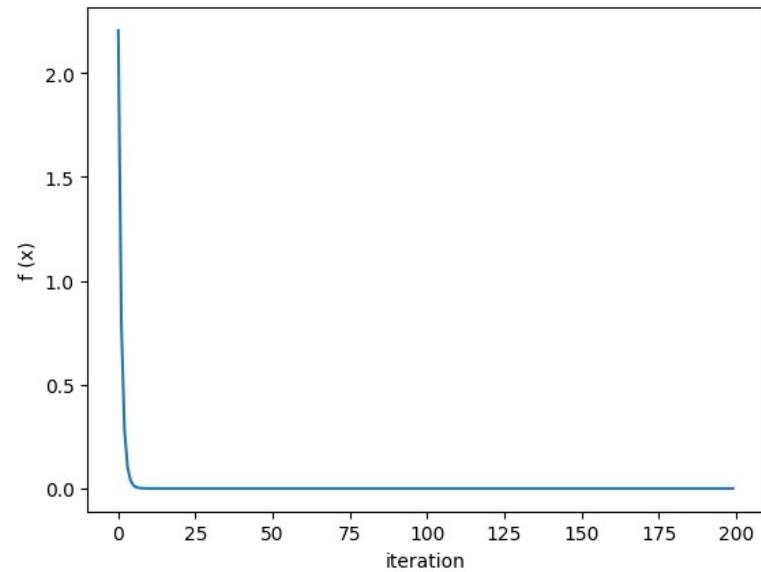
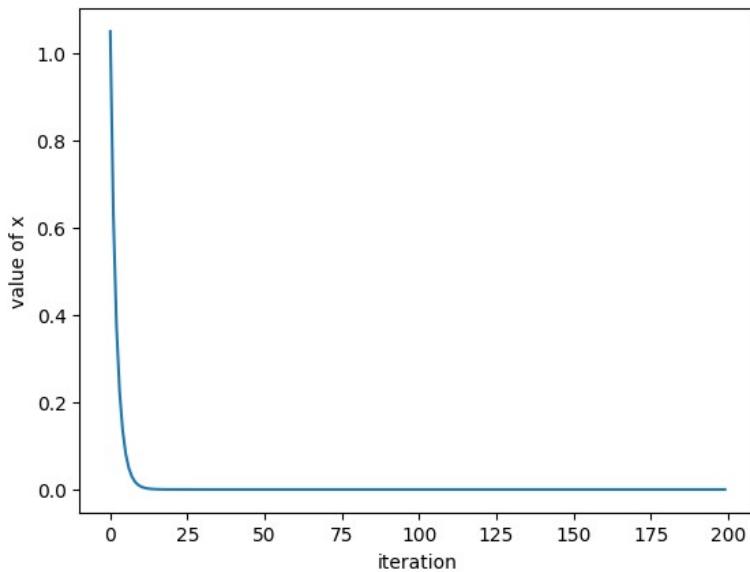
# Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Experiment result (learning\_rate = 0.01)



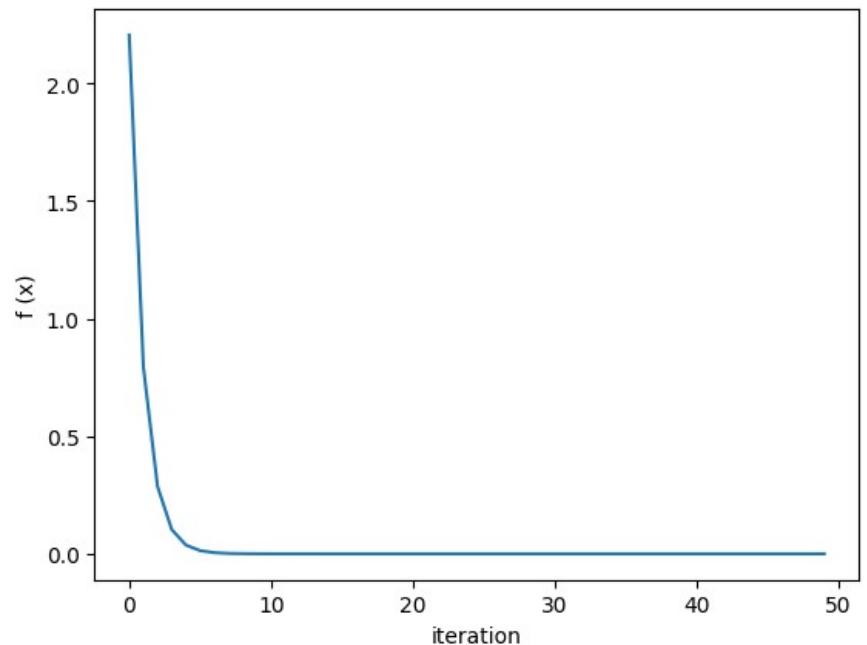
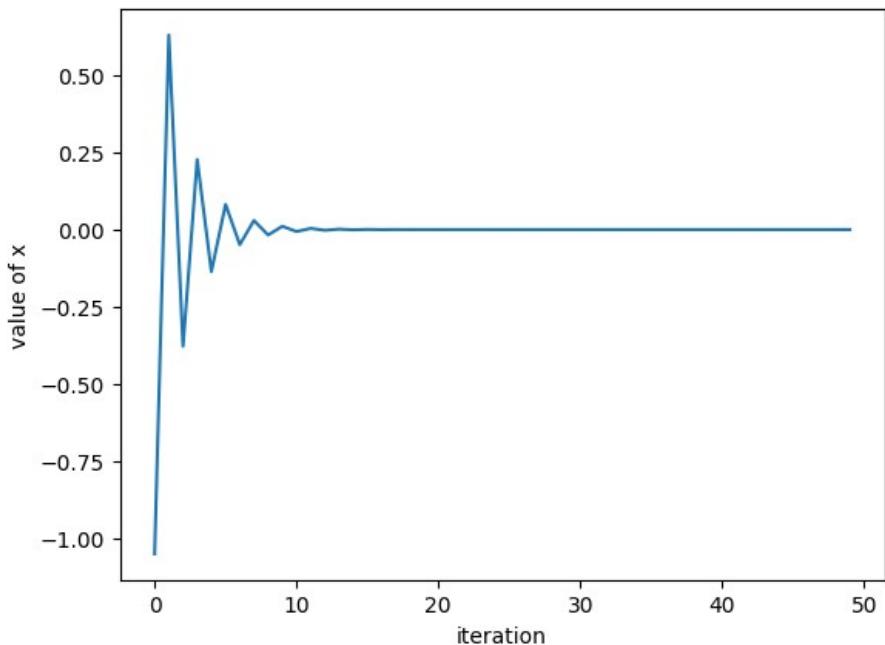
# Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Experiment result (learning\_rate = 0.1)
      - ✓ Lr is increased, converging quicker



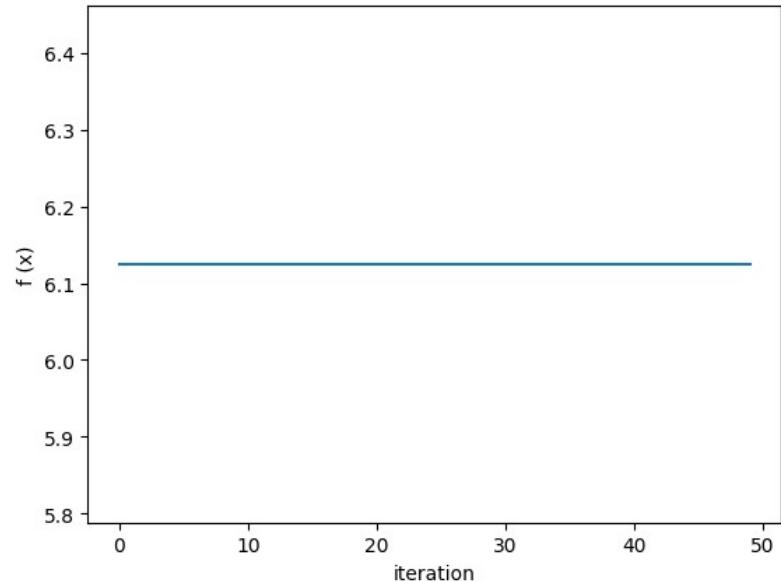
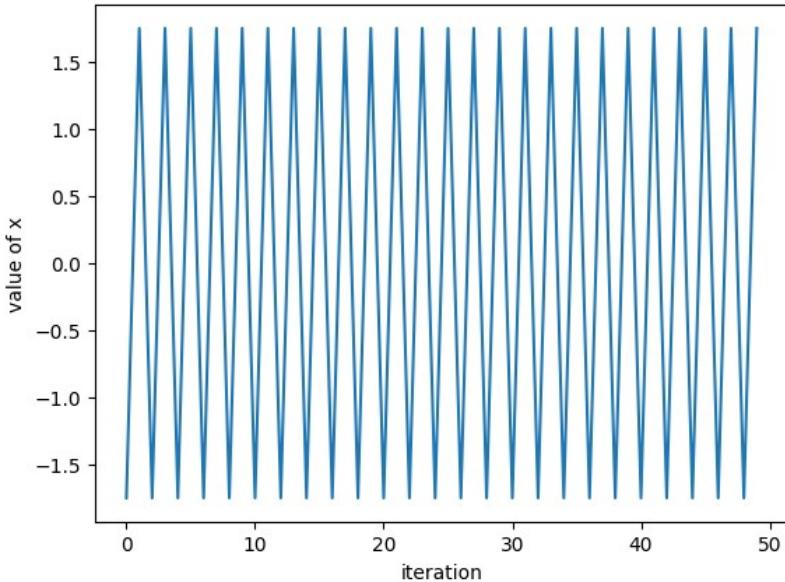
# Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Experiment result (learning\_rate = 0.4, ITER\_MAX=50)
      - ✓ Unstable convergence at the start



# Gradient Descent Algorithm

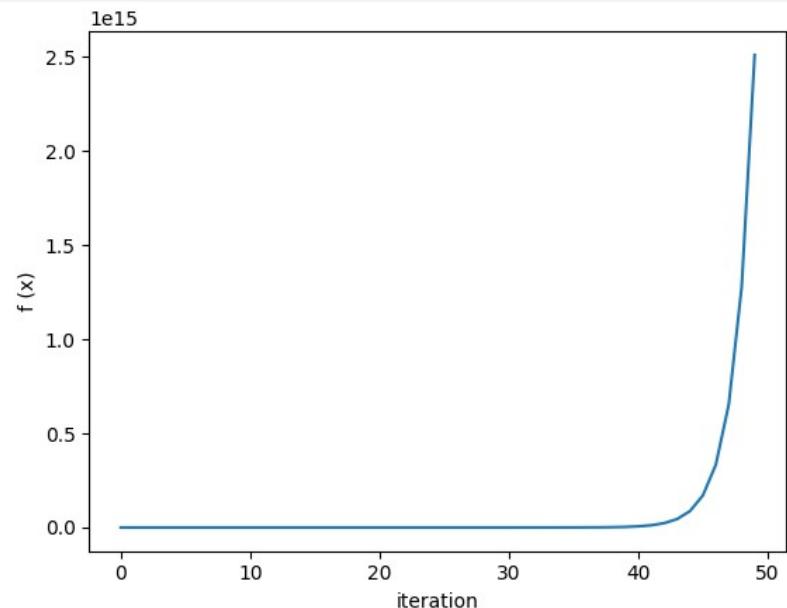
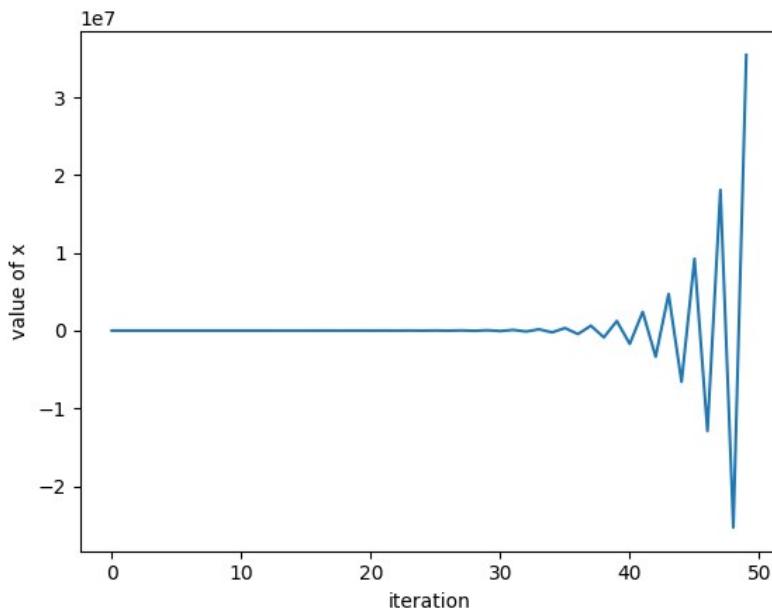
- Python practice 1
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Experiment result (learning\_rate = 0.5, ITER\_MAX=50)
      - ✓ Zig-zag over minimum value



# Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize  $f(x) = 2x^2$ 
    - Experiment result (learning\_rate = 0.6, ITER\_MAX=50)
      - ✓ Diverged, fails to converge

The learning\_rate is too large. As the iterations proceed, the value moves further away from the optimum ( $x = 0$ ). Consequently, the gradient  $\nabla f(x)$  keeps getting larger, which causes the entire step size  $\alpha \times \nabla f(x)$  to continually increase, leading to divergence.



# Gradient Descent Algorithm

- Python practice 2
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Using the derivative value based on the limit definition instead of direct derivative

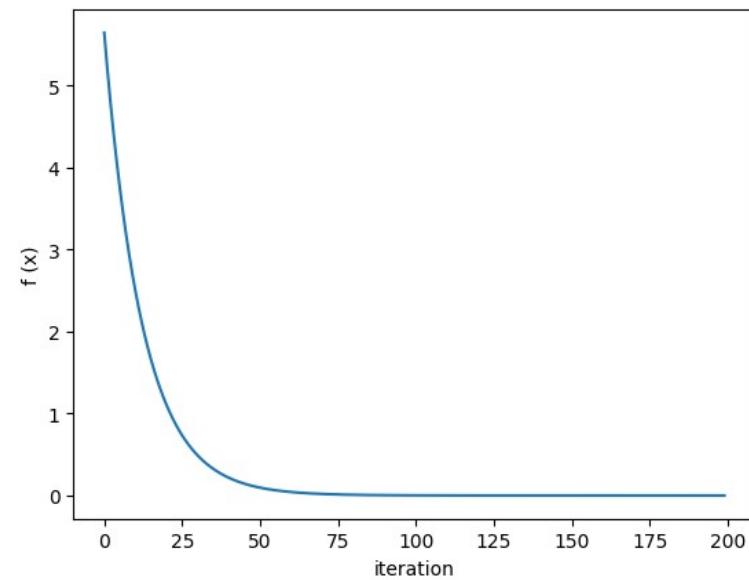
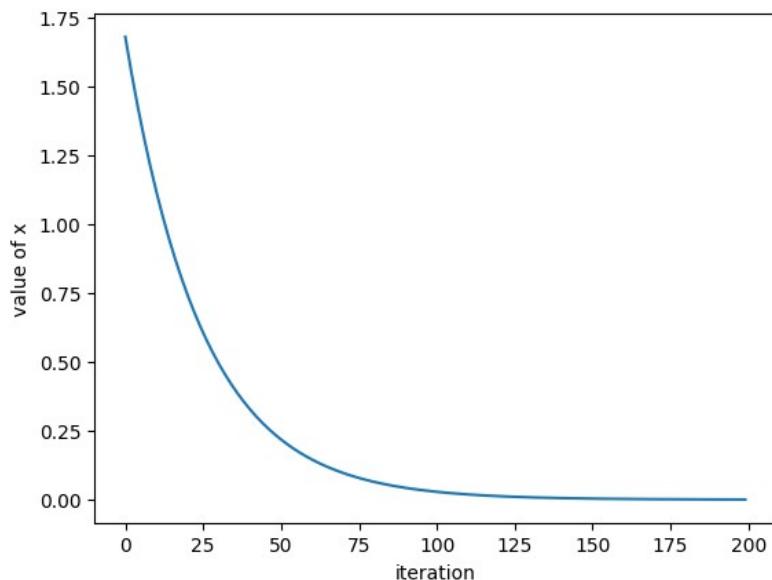
$$\frac{df(x)}{dx} = \lim_{dx \rightarrow 0} \frac{f(x + dx) - f(x)}{dx}$$

```
def f(x): # 목적 함수
    return 2 * (x**2)

def derivative(x): # 목적 함수의 미분
    delta = 0.0001
    return (f(x+delta) - f(x)) / delta
```

# Gradient Descent Algorithm

- Python practice 2
  - Using the GD algorithm to minimize  $f(x)=2x^2$ 
    - Experiment result (learning\_rate = 0.01, ITER\_MAX=200)
      - ✓ **The result is (almost) identical** to the result obtained by calculating and using the analytical derivative.



# Gradient Descent Algorithm

- Python practice 3
  - Using the GD algorithm to minimize  $f(x)=2x_0^2+4(x_1-1)^2$
  - Vector variable  $x = [x_0, x_1]$

```
def f(x): # 목적 함수
    # 2 x0^2 + 4 (x1-1)^2
    return 2 * (x[0]**2) + 4 * ((x[1]-1)**2)

def derivative_x0(x): # 목적 함수의 x0에 대한 미분
    return 4*x[0] + 0

def derivative_x1(x): # 목적 함수의 x1에 대한 미분
    return 0 + 8*(x[1]-1)
```

# Gradient Descent Algorithm

- Python practice 3
  - Using the GD algorithm to minimize  $f(x) = 2x_0^2 + 4(x_1 - 1)^2$
  - Implement GD to take account of the vector variable

```
import random

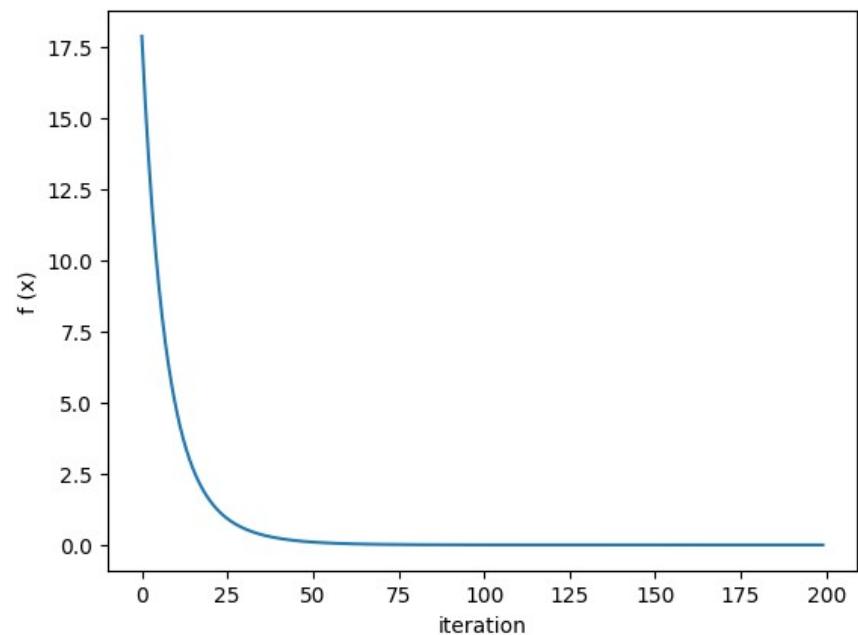
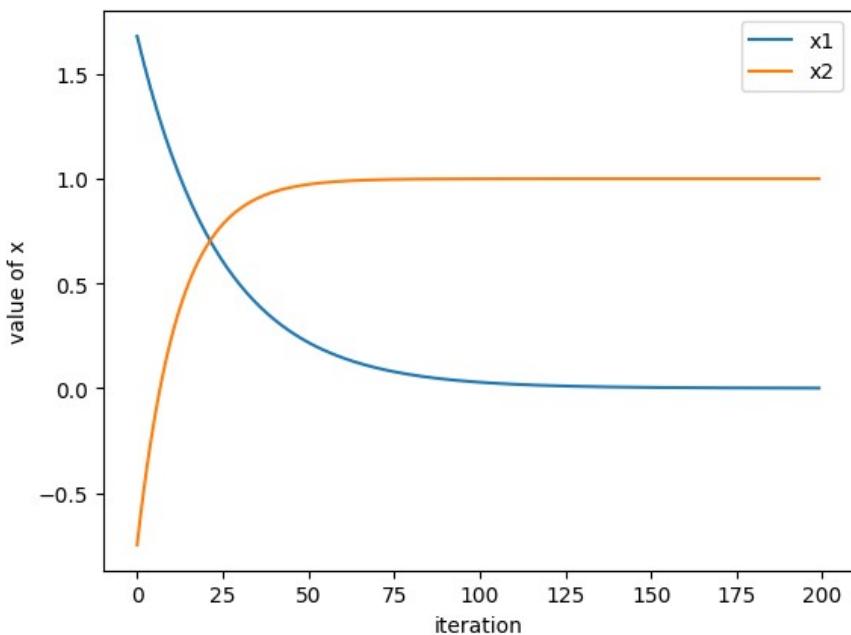
learning_rate = 0.01
#x_current = [random.random()*4 - 2, random.random()*4 - 2] # 무작위 시작점: [-2.0, +2.0)
x_current = [1.75, -0.9] # 공평한 비교 실험을 위해, 시작점을 고정함

def GD(x_current):
    x_new = [x_current[0] - learning_rate * derivative_x0(x_current),
             |   |   | x_current[1] - learning_rate * derivative_x1(x_current)]
    return x_new, f(x_new)

x1s, x2s, fs, ITER_MAX = [], [], [], 200
for i in range(ITER_MAX):
    x_new, f_new = GD(x_current)
    x1s.append(x_new[0])
    x2s.append(x_new[1])
    fs.append(f_new)
    x_current = x_new
```

# Gradient Descent Algorithm

- Python practice 3
  - Using the GD algorithm to minimize  $f(x) = 2x_0^2 + 4(x_1 - 1)^2$
  - Experiment result (learning\_rate = 0.01)



## Adding Gradient Descent (for Numeric Optimization problems)

- Based on the content covered so far, let's learn how to **implement Gradient Descent** for a given **Numerical Optimization problem!**

## Adding Gradient Descent (for Numeric Optimization problems)

- Gradient descent is the same as the steepest-ascent **except the way a next point is created** from the current point
  - Gradient descent generates **only one neighbor**
    - c.f.) Steepest ascent generates  $m$  neighbors from which to select a successor to move to ( $m$  evaluations are needed)
  - Gradient descent computes gradient at the current point and apply the gradient update rule to calculate the next point
    - **$n$  evaluations are needed to calculate partial derivatives in all the dimensions, where  $n$  is the dimension of the objective function (i.e.,  $n =$  number of variables)**
    - **One more evaluation is needed to evaluate the next point** obtained by applying the update rule using the gradient
- Gradient descent is **applicable only to numerical optimization**

## Adding Gradient Descent (for Numeric Optimization problems)

- Two variables are newly added to the `Numeric` subclass:
  - `alpha`: update rate for gradient descent  $x \leftarrow x - \alpha \nabla f(x)$ 
    - Set to a default value of 0.01 for the time being
    - Referenced by the method `takeStep`
  - `dx`: size of the increment used when calculating derivative
    - Set to a default value of  $10^{-4}$  for the time being
    - Referenced by the method `gradient`

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



Simple method for calculating the Gradient:  
Calculate the increase in  $f(x)$  when  $x$  increases infinitesimally.

## Adding Gradient Descent (for Numeric Optimization problems)

- Also, following methods are newly added to the `Numeric` subclass:
  - `takeStep(self, x, v)`:
    - Computes the gradient (`gradient`) of the current point `x` whose objective value is `v`
    - Makes a copy of `x` and changes it to a new one by applying the gradient update rule as long as the new one is within the domain (`isLegal`)

$$\nabla f(x) = \left( \frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \dots, \frac{\partial f(x)}{\partial x_n} \right)^T$$

`isLegal` refers to the case where all individual components  $x_i$  that constitute the vector `x` do not exceed their designated bounds (domain).

$$x_i \leftarrow (x - \alpha \nabla f(x))_i = x_i - \alpha \frac{\partial f(x)}{\partial x_i}$$



if `x_new` is `legal`, return  
`x_new`  
else return `x`

Performs the update for each  $x_i$  and collects them to form `xnew`.

## Adding Gradient Descent (for Numeric Optimization problems)

- **gradient(self, x, v)**
  - o Calculates partial derivatives at  $\mathbf{x}$ 
$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \frac{f(\mathbf{x}') - f(\mathbf{x})}{\delta}$$
$$\mathbf{x}' = (x_1, \dots, x_{i-1}, x_i + \delta, x_{i+1}, \dots, x_d)^T$$
  - o Returns the gradient  $\nabla f(\mathbf{x})$
- **isLegal(self, x)**
  - o Checks if  $\mathbf{x}$  is within the domain for each  $x_i$  in  $\mathbf{x}$
- **getAlpha(self)**
- **getDx(self)**
- **getAlpha** and **getDx** are called from **displaySetting** of the main program when reporting the update rate and the increment size for calculating derivative

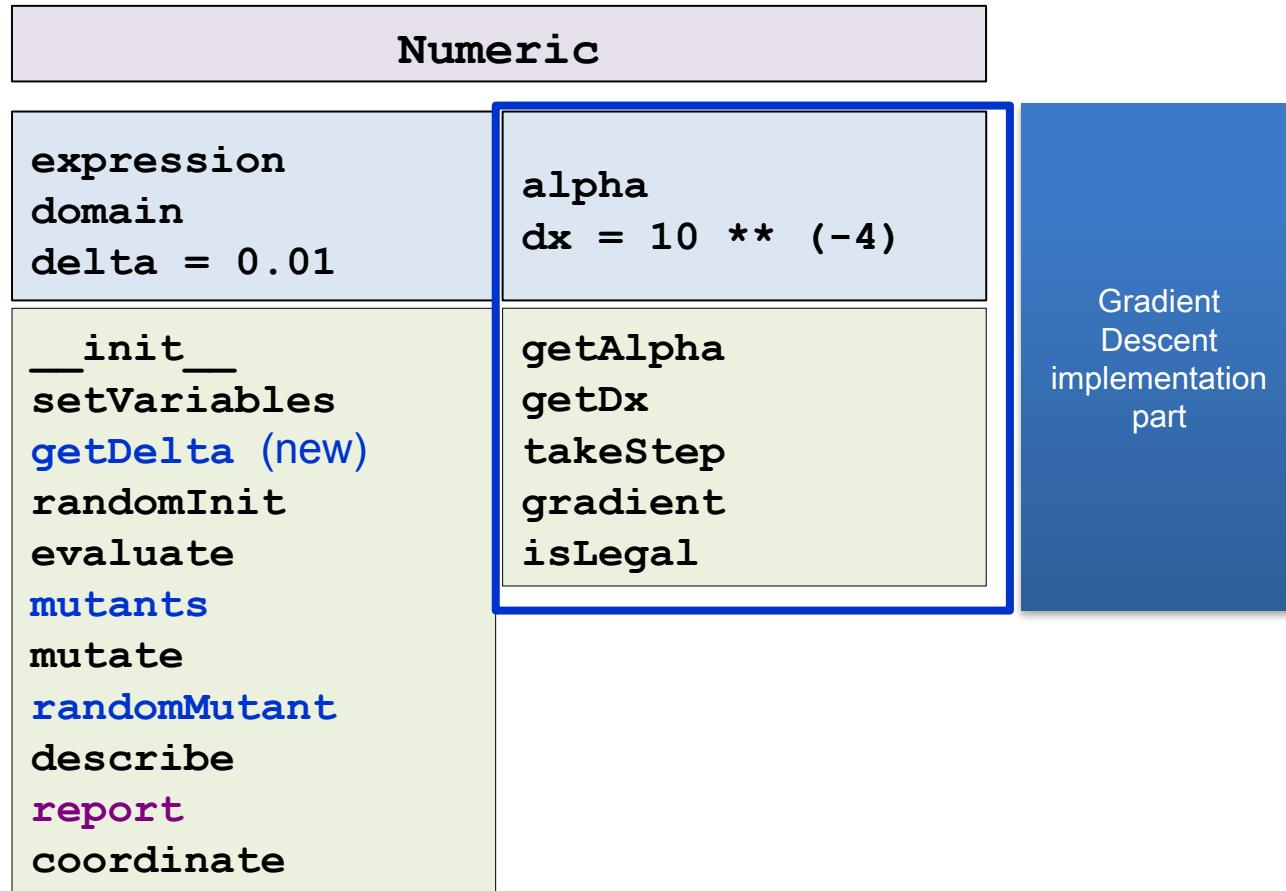
## Adding Gradient Descent (for Numeric Optimization problems)

- Gradient Descent Algorithm (Overall Flow)
  1. Generate a **random starting point** ( $x_{\text{current}}$ ) and calculate its evaluated result ( $v_{\text{current}}$ ).
  2. Generate a new  $x_{\text{new}}$  according to the **GD rule** and calculate its evaluated result ( $v_{\text{new}}$ ).
  3. If  $v_{\text{new}}$  is **better** than  $v_{\text{current}}$ , update  $x_{\text{current}} = x_{\text{new}}$  and  $v_{\text{current}} = v_{\text{new}}$ , and return to **step 2**. Otherwise, terminate the process.
- Numeric.gradient method
  - Calculates the partial derivative  $\frac{\partial f(\mathbf{x})}{\partial x_i}$  for each  $x_i$  and collects them to return the gradient  $\nabla f(\mathbf{x})$ .
- Numeric.takeStep method
  - Uses the  $\frac{\partial f(\mathbf{x})}{\partial x_i}$  calculated by the **Numeric.gradient method** to perform the **update for each**  $x_i$ , and collects them to return  $x_{\text{new}}$ .

variables

functions

# Final Class Hierarchy



# GD Implementation Code File

```
gradient_descent.py
```

```
def main():
    p = Numeric()
    ...
def gradientDescent(p):
    ...
def displaySetting(p):
    ...

main()
```

```
def main():
    # Create a Problem object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables()   # Set its class variables (expression, domain)
    # Call the search algorithm
    gradientDescent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

# Search Algorithms: Object-Oriented Implementation (Part D)

# Contents

- Conventional vs. AI Algorithms
- Local Search Algorithms
- Implementing Hill-Climbing Algorithms
- Defining ‘Problem’ Class
- Adding Gradient Descent
- **Defining ‘HillClimbing’ Class**
- Adding More Algorithms and Classes
- Experiments

Goal: Implement a `HillClimbing` class that integrates multiple Search Algorithms into a single unit!

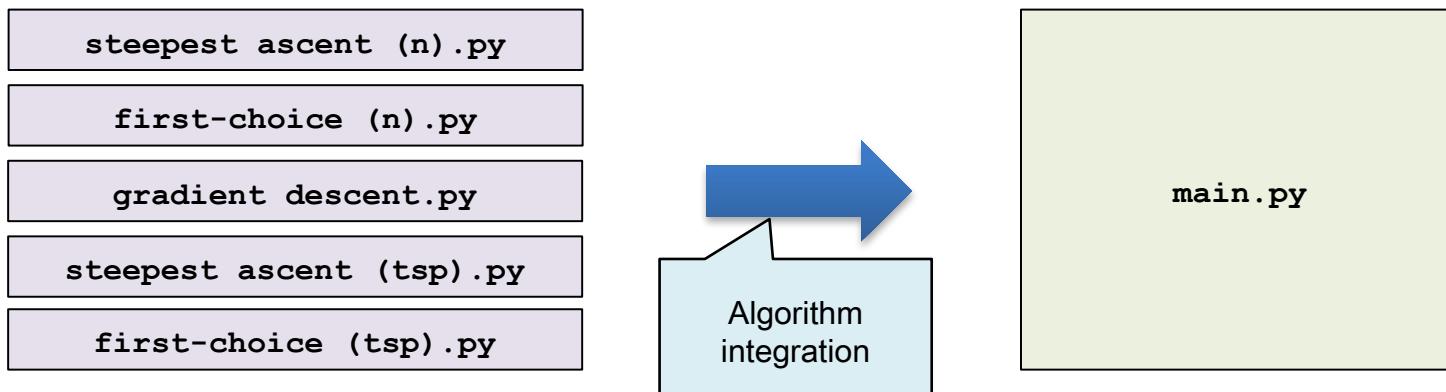
# Summary of Implementation So Far

- Search Algorithm (C) / Search Tool v2 Implementation Status
  - problem.py : Definition of the class for *variables/functions related to the problem's solution.*
    - Problem Class definition : Includes content common regardless of the problem type (e.g., saving the solution, saving numEval, etc.)
    - Defines two subclasses based on the problem type, and defines information and frequently used functions related to each problem:
      - ✓ Numeric Class (*inheriting from Problem*)
      - ✓ TSP Class (*inheriting from Problem*)
  - And, the solution algorithms for solving problems are each implemented in **separate files**.
    - Separate source code and main function are defined according to the solution algorithm used to solve the problem
    - Implementation code for solution algorithms to solve **Numerical Optimization** problems:
      - ✓ steepest ascent (n).py : Code to solve *Numeric* problems using the **SA** method
      - ✓ first-choice (n).py : Code to solve *Numeric* problems using the **FC** method
      - ✓ gradient descent.py : Code to solve *Numeric* problems using the **GD** method
    - Implementation code for solution algorithms to solve **TSP** problems:
      - ✓ steepest ascent (tsp).py : Code to solve *TSP* problems using the **SA** method
      - ✓ first-choice (tsp).py : Code to solve *TSP* problems using the **FC** method

Today's  
Topic

# Code Improvement Idea

- Search Algorithm (C) - Search Tool v2 drawbacks
  - Each solution algorithm is implemented with a **separate main program** (source code).
  - To run a specific algorithm, **the corresponding program must be found and executed**.
  - If a new solution algorithm is added, a **new program and new source code** are required.
    - Ex: To add the Gradient Descent algorithm, a new `gradient_descent.py` source code was created and implemented.
- Idea for Improvement
  - **Integrate** the implemented solution algorithms from *different source codes* into **one main program** (= one source code) and allow the user to select the solution algorithm during program execution!



|   |
|---|
|  main.py       |
|  optimizer.py |
|  problem.py  |
|  setup.py    |

# Code Improvement Idea

- Search Algorithm (C) Search Tool v2: Implementation Method to [v3](#)
  - Source code structure:
    - `main.py` : Receives *user input*, selects the **Problem Type** and the **Solution Algorithm**
    - `problem.py` : Code definition related to the Numeric and TSP problems.
    - `optimizer.py` : Definition of the *classes and methods (functions)* for the solution algorithm implementation.
    - `setup.py` : Definition of the functions and management of information commonly used by the Problem class and the HillClimbing class.
  - Class structure change:
    - Extract the common information/attributes used by the existing Problem class and the newly defined HillClimbing class, define them in a **separate Setup class**, and modify the code so that both the Problem class and the HillClimbing class inherit from it.
  - Class for solution algorithms : `optimizer.py`
    - Implement the class HillClimbing, and define the respective SteepestAscent, FirstChoice, and GradientDescent classes that *inherit* from it.
  - Main program: `main.py`
    - Write a single main program and allow the user to select simultaneously via user input:
      - ✓ (1) Which problem to solve (Numeric or TSP), and
      - ✓ (2) Which solution algorithm to use (SA, FC, GD).

# Search Tool v3

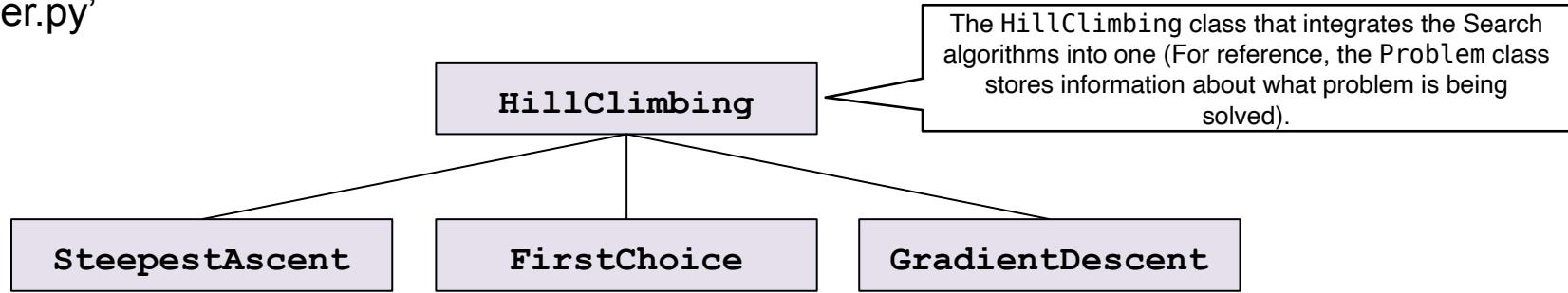
- Search Algorithm (D) / Search Tool v3 Implementation details
  - Implementation of the top-level Setup class to store delta, alpha, and dx values.
    - The Problem class (for problems) and the HillClimbing class (for solution algorithms) will both inherit from the Setup class.
    - delta : Used when generating mutants (= neighbor solution) in the Numeric Optimization problem type.
    - alpha : Determines the *step size* or *learning rate* in the **Gradient Descent** solution algorithm.
    - dx : Used when calculating the *gradient* in the **Gradient Descent** solution algorithm.
  - problem.py : Defines and differentiates classes based on the **nature of the problem**.
    - Definition of the Problem class (inheriting from Setup).
    - Definition of the Numeric class (inheriting from Problem).
    - Definition of the TSP class (inheriting from Problem).

# Search Tool v3

- Search Algorithm (D) / Search Tool v3 Implementation details
  - `optimizer.py` : Defines and differentiates classes according to the **solution algorithm used to solve the problem**
    - Definition of the HillClimbing class (inheriting from Setup).
    - Defines the SteepestAscent class (inheriting from HillClimbing) and implements the class using the code previously implemented in `steepest ascent (n.tsp).py`.
    - Defines the FirstChoice class (inheriting from HillClimbing) and implements the class using the code previously implemented in `first-choice (n.tsp).py`.
    - Defines the GradientDescent class (inheriting from HillClimbing) and implements the class using the code previously implemented in `gradient descent.py`.
  - `main.py`: Implements the user interface (UI) so the user can make selections during program execution.
    - Select which type of problem to solve: Numeric or TSP class.
    - Select which solution algorithm to use for the chosen problem: SteepestAscent, FirstChoice, or GradientDescent class.
    - **Note:** Logic is required to prevent the user from selecting the Gradient Descent algorithm in the case of a TSP problem.

# Defining ‘HillClimbing’ Class

- We define `Hillclimbing` class to put together all the search algorithms (`optimizer.py`) and unite all the programs into a single main program (`main.py`)
  - Search algorithms become subclasses under `Hillclimbing`
  - The class hierarchy of `Hillclimbing` is stored in a separate file named ‘optimizer.py’



- The names of the search algorithms are now the names of the subclasses under the `Hillclimbing` class
  - The body of each search algorithm becomes the body of the `run` method of the corresponding subclass

# Defining ‘HillClimbing’ Class

- To have a single main program, we need a new user interface to ask the user the type of problem to be solved (pType) and the type of algorithm to be used (aType) [pType, aType is `int` datatype]
  - These information will be used to create the `Problem` (= Numeric or Tsp) and `HillClimbing` (= SteepestAscent, FirstChoice or GradientDescent) objects of the right types
  - `pType` will also be used when printing out messages about the settings of the search algorithm used
- `displaySetting` that was previously part of the main program is moved to the `HillClimbing` class
  - because what it displays are the information about the settings of the search algorithms that are now the methods of `HillClimbing`

# Defining ‘HillClimbing’ Class

- To store information necessary for `displaySetting` and the search algorithms, two variables are defined in `HillClimbing`
  - `pType`: integer indicating the `type of problem` to be solved
  - `limitStuck`: maximum evaluations allowed without improvement
    - Previously, it was `LIMIT_STUCK`
    - Currently, only `firstChoice` is under the control of this variable
    - Later, the stochastic hill-climbing algorithm to be added to the search tool will be controlled by this variable
- `displaySetting` in `HillClimbing` prints out the mutation step size (`delta`) when the type of problem is numerical optimization
  - This method is inherited to `steepestAscent` and `FirstChoice`, but not to `GradientDescent`

# Defining ‘HillClimbing’ Class

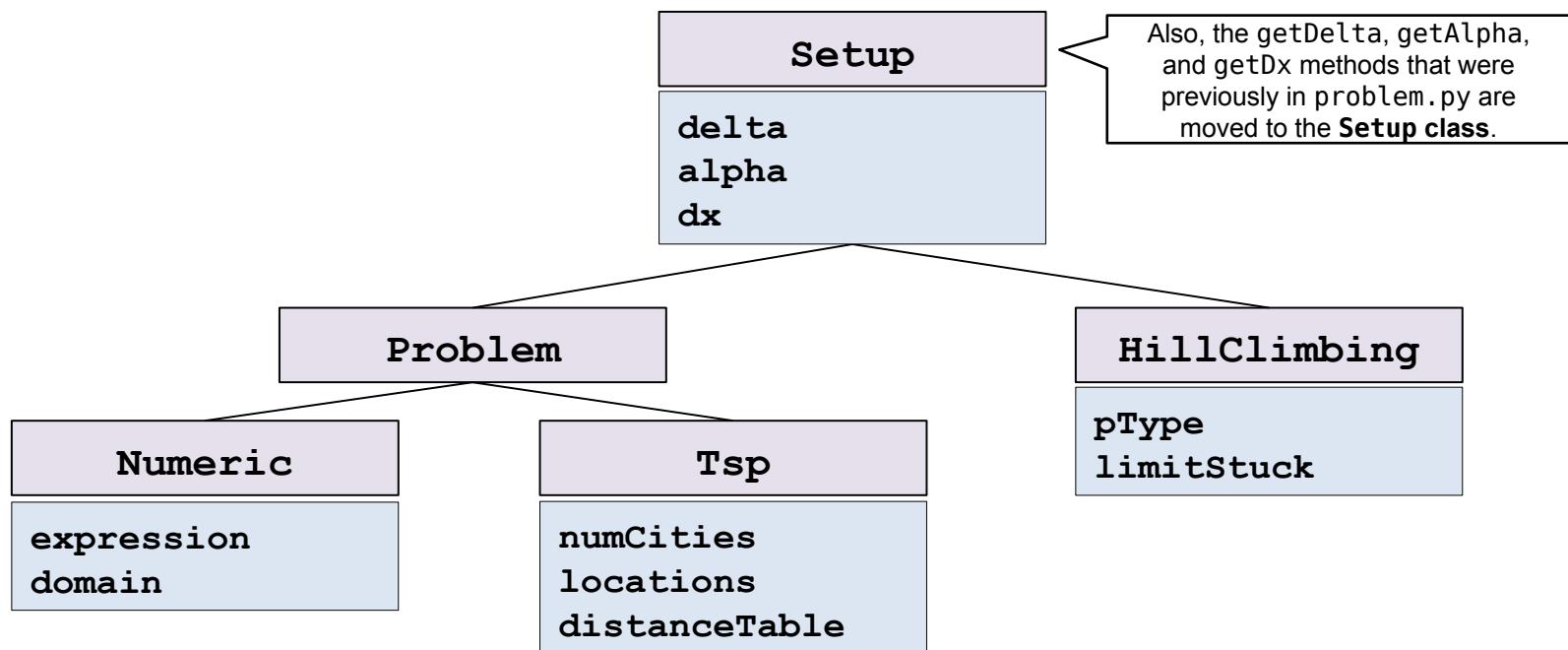
- `displaySetting` in each class of search algorithm prints out the algorithm name and additional setting information specific to that algorithm
  - `displaySetting` of `FirstChoice` prints out the maximum evaluations allowed without improvement (`limitStuck`)
  - `displaySetting` of `GradientDescent` prints out the values of `alpha` and `dx` (using accessor methods `getAlpha` / `getDx` or they can be directly accessed via `self._alpha` / `self._dx`)
- Notice that `delta`, `alpha`, `dx` were variables of `Numeric` subclass (under `Problem`), and `getDelta`, `getAlpha`, `getDx` were methods of `Numeric`
  - For `displaySetting` to refer to `alpha` and `dx`, it needs as its argument the problem instance being solved (so that it can use the statement such as `p.getAlpha()`)
  - 다음 페이지 계속 ...

# Defining ‘HillClimbing’ Class

- What if we make `alpha` and `dx` variables of `GradientDescent`, and `delta` a variable of `HillClimbing`?
  - Not a good idea because they are already variables of the subclass `Numeric`
- We better create a superclass of `Numeric` and `HillClimbing`, and have those variables belong to that superclass
  - Once this is done, the three accessors, `getDelta`, `getAlpha`, and `getDx` of the `Problem` class are no longer necessary because `displaySetting` that deals with these information will belong to `HillClimbing` and thus can access those variables directly
  - 다음 페이지 계속 ...

# Adding a Superclass ‘Setup’

- Since `delta`, `alpha`, and `dx` are needed by both the classes `Hillclimbing` and `Problem`, we define a new class named `Setup` to hold those variables and make it a parent class of both
- We store `Setup` in a separate file named ‘`setup.py`’ and let the ‘`problem.py`’ and ‘`optimizer.py`’ files import it from that file



# The Main Program

- The main program is stored in a file named ‘main.py’
  - The ‘main.py’ file should import everything from ‘problem.py’ and ‘optimizer.py’
- The main program includes a few functions for a new **user interface** to ask the user to choose the problem to be solved and the optimization algorithm to be used

Receive aType information as input to determine which **algorithm** to use (Steepest Ascent or First Choice or Gradient Descent).

Receive pType information as input to determine which type of **problem** to solve (Numeric Opt or TSP).

# The Main Program

- `main.py` is comprised of the following four functions:
- 1. `main()`:
  - Creates a `Problem` object `p` of the right type by querying to the user (`selectProblem`)
  - Creates a `Hillclimbing` object `alg` (search algorithm) by querying to the user (`selectAlgorithm`)
  - Runs the search algorithm by calling the `run` method of the `Hillclimbing` class (`alg.run`)
  - Shows the specifics of the problem just solved (`p.describe`)
  - Shows the settings of the search algorithm (`alg.displaySettings`)
  - Displays the result of search (`p.report`)

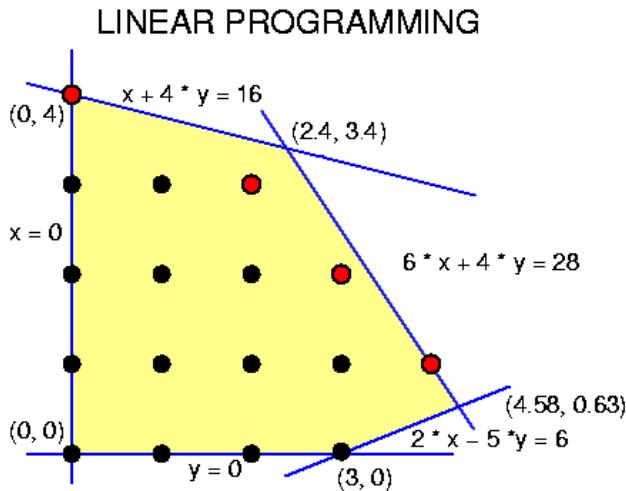
# The Main Program

- 2. **selectProblem()**:
  - Asks the user to choose **the type of problem to be solved**
  - Creates a **Problem** object **p** of the right type
  - Sets the variables of the corresponding subclasses of **Problem**
  - Returns **p** and **pType** (an integer indicating the problem type)
- 3. **selectAlgorithm(pType)**:
  - Asks the user to **select a search algorithm**
  - Asks the user to select again if gradient descent is chosen for a TSP (**invalid**)
  - Prepares a **dictionary** whose keys are integers corresponding to **aType**, and values are the names of the subclasses of **HillClimbing** (i.e., search algorithms)
  - Creates an object **alg** of the targeted **HillClimbing** subclass using the dictionary (**alg = eval(optimizers[aType])**)
  - Sets the variables of the **HillClimbing** class
  - Returns **alg**

# The Main Program

- 4. `invalid(pType, aType)`:
  - If gradient descent is chosen for a TSP (= 잘못된 선택), informs the fact to the user and returns `True`
  - Otherwise, returns `False`

The **Gradient Descent** algorithm can only solve problems of the Continuous Space type and **cannot be used** for Discrete type TSP problems.



Function to maximize:  $f(x, y) = 6 * x + 5 * y$

Optimum LP solution  $(x, y) = (2.4, 3.4)$

Pareto optima:  $(0, 4), (2, 3), (3, 2), (4, 1)$

Optimum ILP solution  $(x, y) = (4, 1)$

- Continuous problem
  - Every point within a certain range can be a solution.
  - Ex: In a problem that minimizes  $x^2 + y^2$ , the following ranges of values are valid and can be used as solutions:  $-1 \leq x \leq 1, -3 \leq y \leq 7$ .
- Discrete Optimization problem
  - Only specific, distinct values can be solutions.
  - Ex: Which order should we visit Seoul and Busan?  $x = 1$  (Seoul → Busan), 2 (Busan → Seoul)
- When the Gradient or its opposite direction is used to find the next solution (next solution) based on the current solution (current solution), finding a valid solution in this manner for a **Discrete** case

# Order of operation

- main.py > main()
  - Call **selectProblem()**
    - Receive Problem Type input form the user (1=Numeric, 2=TSP) => store as **pType**
    - Based on pType, create Numeric or TSP class instance => store in variable **p**
    - Call **p.setVariables**
      - ✓ Receive the file name where the problem is stored from the user.
      - ✓ Read the stored file, extract the **problem information**, and store it in the class's instance variables.
  - Call **selectAlgorithm(pType)**
    - Receive input from the user for which **algorithm** to use (1: SA, 2: FC, 3: GD) => store as **aType**
    - Call the **invalid** function to check (e.g. TSP cannot be solved by GD)
    - Create a class instance corresponding to the solution algorithm entered by the user => store in variable **alg**
  - **alg.run(p)** to execute solution algorithm
    - The implementation of **alg.run** is \_identical to the content of the solution algorithm code\_ previously implemented in the separate files.
  - Output problem information, parameter information, and execution results (solution, etc.) to the screen by sequentially calling the functions: **p.describe()** , **alg.displaySettings()** , **p.report()** .

# Source Code (summary)

- Source code structure
  - main.py : Receives **user input**, selects the problem type and solution algorithm, and executes the algorithm.
    - `main`, `selectProblem`, `selectAlgorithm`, `invalid` functions.
  - problem.py : Defines code related to Numeric and TSP problems.
    - Implementation: `Problem` class, `Numeric` class, `TSP` class.
  - optimizer.py : Implements classes and methods (functions) for the solution algorithm implementation.
    - `HillClimbing` Class: `setVariables`, `displaySetting`, `run` (pass) methods
    - `SteepestAscent` Class: `displaySetting`, `run`, `bestof` methods
    - `FirstChoice` Class: `displaySetting`, `run` methods
    - `GradientDescent` Class: `displaySetting`, `run` methods
    - Implementation is the same as the solution algorithm code previously in separate files.
  - setup.py : Defines functions and manages information commonly used by the Problem class and the HillClimbing class.
    - Implementation: `getDelta`, `getAlpha`, `getDx`

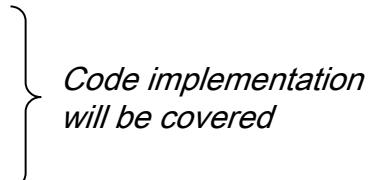
# Search Algorithms: Object-Oriented Implementation (Part E)

# 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
- **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).

Already implemented

# Review (2/2)

- From 09 Search Algorithm (A) previously...

- Simulated Annealing Search Algorithm:

- 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*  $\leftarrow$  MAKE-NODE(*problem.INITIAL-STATE*)

Calculate the current state from the initial state

**for** *t*  $\leftarrow 1$  **to**  $\infty$  **do**

*T*  $\leftarrow$  *schedule[t]*

**if** *T* = 0 **then return** *current*

*next*  $\leftarrow$  a randomly selected successor of *current*

Randomly select the next state

$\Delta E \leftarrow$  *next.VALUE* - *current.VALUE*

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

**if**  $\Delta E < 0$  **then** *current*  $\leftarrow$  *next*

Assumes a minimization problem  $\rightarrow$  Lower value is better

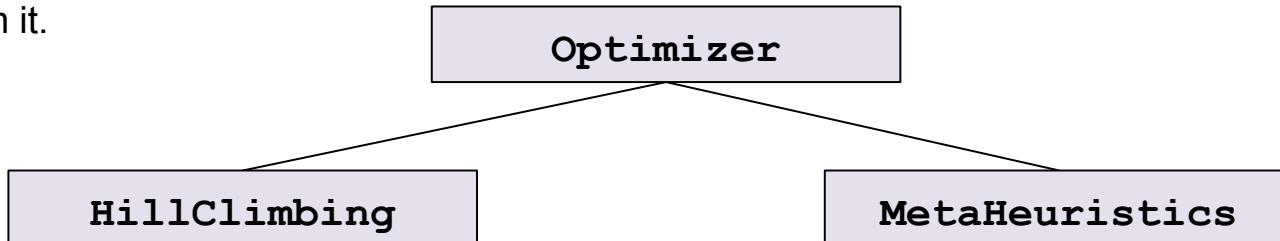
**else** *current*  $\leftarrow$  *next* only with probability  $e^{-\Delta E/T}$

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

New algorithm to be implemented

# 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.

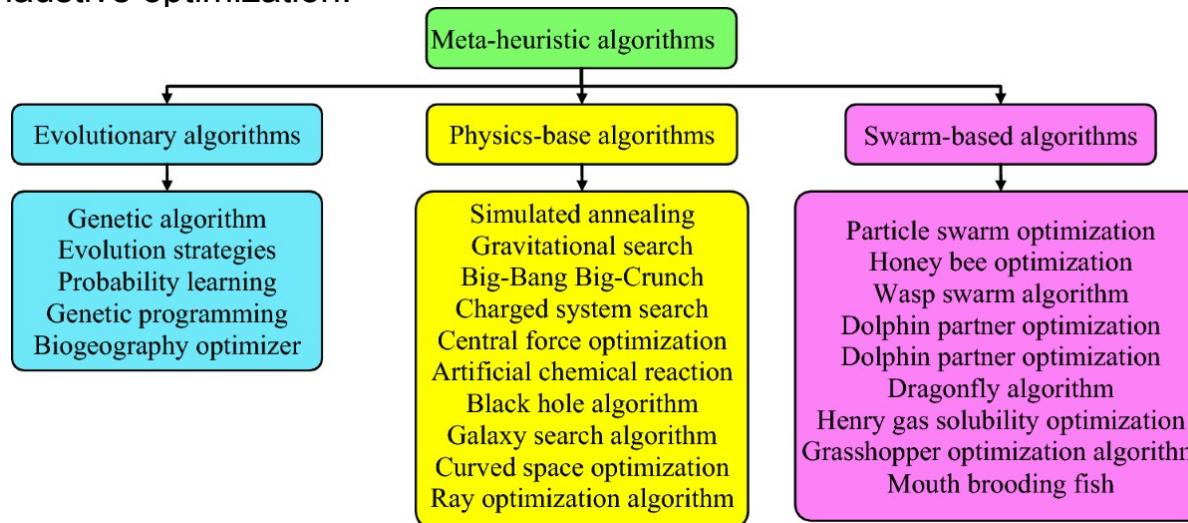


# Adding More Algorithms and Classes

- (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.

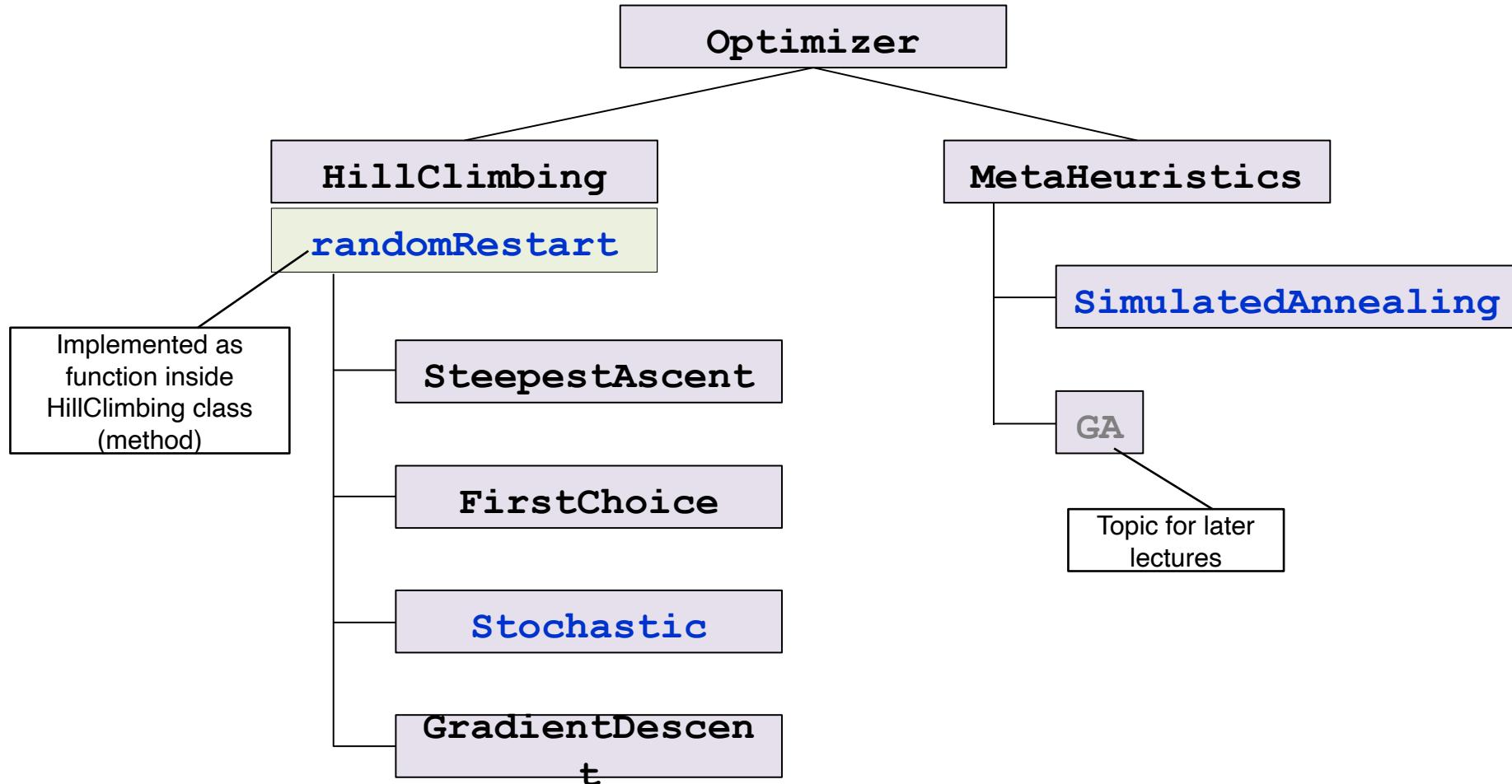
# Adding More Algorithms and Classes

- (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

optimizer.py : Modified configuration



# 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: SteepestAscent")  
        print()  
        HillClimbing.displaySetting(self)
```

```
def run(self, p):
```

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

# 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):

```
2 # Select the problem type:  
3 #   1. Numerical optimization  
4 #   2. TSP  
5 | Enter the number (pType) : 2  
6 #  
7 #   Enter the name of the file : problem/Convex.txt  
8 #   Enter the name of the file : problem/Griewank.txt  
9 #   Enter the name of the file : problem/Ackley.txt  
10 #  Enter the name of the file : problem/tsp30.txt  
11 #  Enter the name of the file : problem/tsp50.txt  
12 | Enter the name of the file : problem/tsp100.txt  
13 #  
14 # Select the search algorithm:  
15 # Hill Climbing algorithms:  
16 #   1. Steepest-Ascent  
17 #   2. First-Choice  
18 #   3. Stochastic  
19 #   4. Gradient Descent  
20 # Metaheuristic algorithms:  
21 #   5. Simulated Annealing  
22 | Enter the number (aType) : 2  
23 #
```

```
24 # If you are solving a function optimization problem,  
25 # what should be the step size for axis-parallel mutation?  
26 | Mutation step size (delta) : 0.01  
27 #  
28 # If your algorithm choice is 2 or 3,  
29 # what should be the number of consecutive iterations without improvement?  
30 | Give the number of iterations (limitStuck) : 1000  
31 #  
32 # If your algorithm choice is 4 (gradient descent),  
33 # what should be the update step size and increment for derivative?  
34 | Update rate for gradient descent (alpha) : 0.01  
35 | Increment for calculating derivative (dx) : 10 ** (-4)  
36 #  
37 # If you want a random-restart hill climbing,  
38 # enter the number of restart.  
39 # Enter 1 if you do not want a random-restart.  
40 | Number of restarts (numRestart) : 10  
41 #  
42 # If you are running a metaheuristic algorithm,  
43 # what should be the total number of evaluations until termination?  
44 | Enter the number (limitEval) : 100000  
45 #  
46 # Enter the total number of experiments  
47 | 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
  - 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...

Added Details

```
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
    :
    :

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

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.

Temperature scheduling  
setup

Search Algorithms: Object-Oriented Implementation (Part E)

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

Annotation (Part E)

# 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']

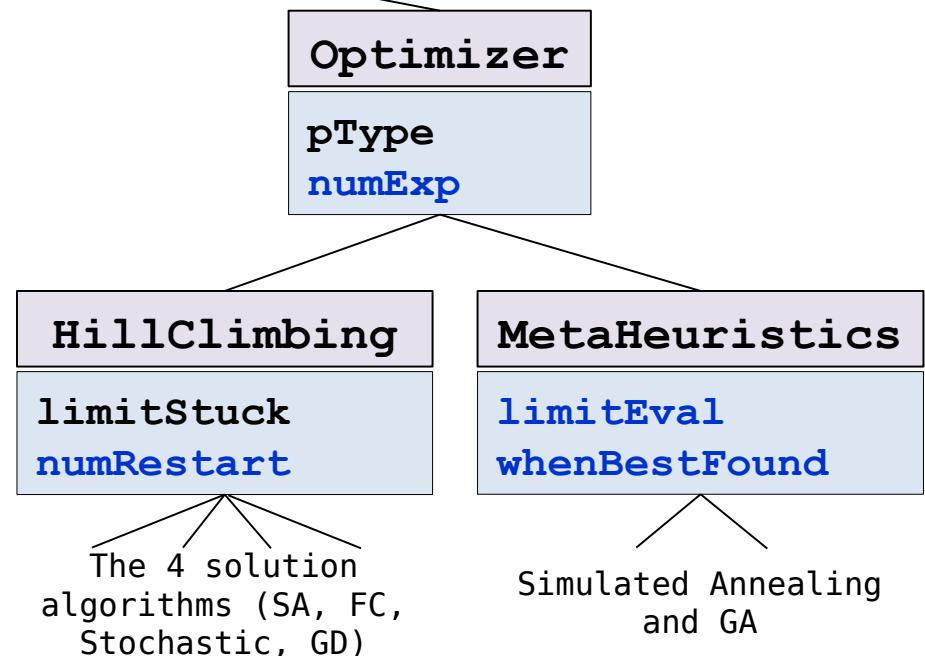
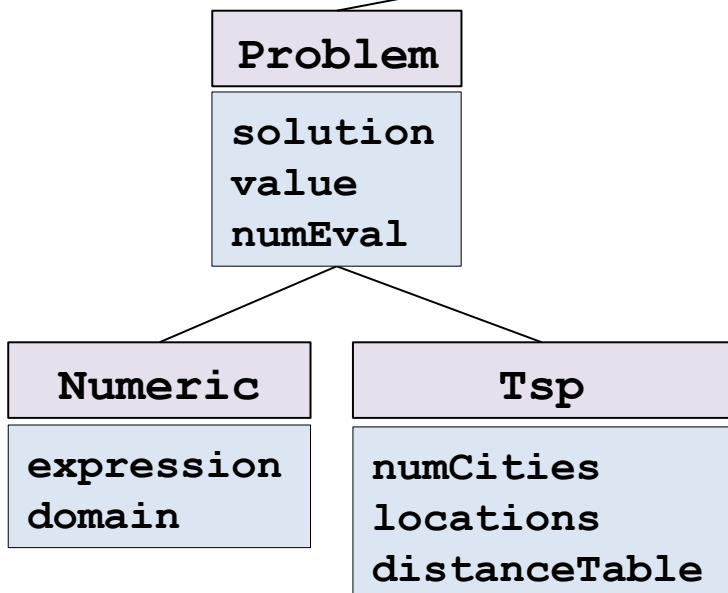
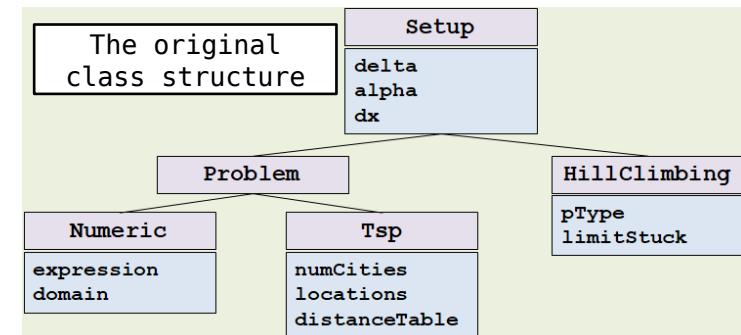
    def getNumExp(self):
        return self._numExp

    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.



# 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)`

```
class HillClimbing(Optimizer):  
    def __init__(self):  
        Optimizer.__init__(self)  
        self._limitStuck = 0 # Max evaluations allowed  
        self._numRestart = 0 # Number of restarts  
  
    def setVariables(self, parameters):  
        Optimizer.setVariables(self, parameters)  
        self._limitStuck = parameters['limitStuck']  
        self._numRestart = parameters['numRestart']
```

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

```
class Setup:  
    def __init__(self):  
        self._aType = 0 # Type of optimizer  
        self._delta = 0 # Step size for axis  
        self._alpha = 0 # Update rate for gradient  
        self._dx = 0 # Increment for calculating gradients  
  
    def setVariables(self, parameters):  
        self._aType = parameters['aType']  
        self._delta = parameters['delta']  
        self._alpha = parameters['alpha']  
        self._dx = parameters['dx']
```

# 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.

```
class HillClimbing(Optimizer):  
    def __init__(self):  
        Optimizer.__init__(self)  
        self._limitStuck = 0 # Max evaluations allowed  
        self._numRestart = 0 # Number of restarts
```

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

```
class Setup:  
    def __init__(self):  
        self._aType = 0 # Type of optimizer  
        self._delta = 0 # Step size for axis  
        self._alpha = 0 # Update rate for gradient  
        self._dx = 0 # Increment for calculating gradients
```

# Source Code Modification Summary

| Code         | Change details   |
|--------------|--|
| setup.py     | <ul style="list-style-type: none"><li>The <i>aType</i> variable was added, and the <i>setVariables</i> method was added.</li></ul>   |
| problem.py   | <ul style="list-style-type: none"><li>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.</li></ul>   |
| main.py      | <ul style="list-style-type: none"><li>The way information like <b>problem type/algorithm type</b> is passed has changed. There is a major change in the user interface, such as reading variables (e.g., <i>dx</i>) that were previously fixed as default values from a text file, leading to overall code modification.</li></ul> |
| optimizer.py | <ul style="list-style-type: none"><li>A new superclass called <i>Optimizer</i> was added.</li><li>Due to the addition of 3 solution algorithms (<i>random</i>, <i>stochastic</i>, <i>simulated annealing</i>), new classes were defined: <i>Stochastic</i>, <i>MetaHeuristics</i>, <i>SimulatedAnnealing</i>.</li></ul>            |

# Source Code Structure

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

| Method  | Description   |
|---|---|
| <code>readPlanAndCreate</code> ,<br><code>readValidPlan</code> ,<br><code>readPlan</code> | <ul style="list-style-type: none"><li>• Obtains information related to the problem and algorithm execution from the file (at this time, lines treated as comments in the file are skipped).</li><li>• Creates separate <b>class instances</b> corresponding to the problem and the algorithm.</li></ul> |
| <code>conductExperiment</code>  | <ul style="list-style-type: none"><li>• Executes the algorithm to <b>find a solution</b> for the given problem.</li></ul>   |

# Source Code Structure

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

| Method                                  | Description  |
|---|--|
| <code>HillClimbing.randomRestart</code> | <ul style="list-style-type: none"><li>• Executes the <code>run</code> method for <code>numRestart</code> times and saves the best solution among them (<code>storeResult</code>)</li><li>• <b>Note:</b> The <code>run</code> method starts by generating a <b>random initial solution</b> and repeats until a better solution no longer emerges.</li></ul> |
| <code>Stochastic.run</code>             | <ul style="list-style-type: none"><li>• Similar to <i>steepest ascent</i>, it generates <b>neighbor solutions</b> randomly, then <b>probabilistically selects</b> the best solution among them (<code>stochasticBest</code>).</li></ul>  |
| <code>Stochastic.stochasticBest</code>  | <ul style="list-style-type: none"><li>• <b>Probabilistically selects one</b> from among the neighbor solutions (at this time, it uses the results of evaluating each solution).</li></ul>  |

# Source Code Structure

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

| Method                                    | Description   |
|---|---|
| <code>SimulatedAnnealing.run</code>       | <ul style="list-style-type: none"><li>• Generates <b>one random neighbor solution</b> (similar to <i>FirstChoice</i>).</li><li>• Selects the neighbor solution <b>probabilistically</b>.</li></ul>  |
| <code>SimulatedAnnealing.initTemp</code>  | <ul style="list-style-type: none"><li>• Sets the initial parameters so that the <b>probability of selecting a bad neighbor solution is 0.5</b> (Initial temperature setting).</li></ul>   |
| <code>SimulatedAnnealing.tSchedule</code> | <ul style="list-style-type: none"><li>• <b>Gradually decreases the probability</b> of selecting a bad neighbor solution as the <i>iteration</i> repeats.</li><li>• That is, it is gradually decreased by the temperature value (<i>temperature</i> is high -&gt; high probability of selecting a bad solution).</li></ul> |

# 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
- (TODO) Also, modify the **class structure** for expandability/extensibility
  - To easily add various types of algorithms
- (TODO) Add configuration files
  - Use a configuration file where the problem to be solved, the solution algorithm, and various parameters are predefined
  - As a result, the code that interacts with the user was modified, and logic for parsing and processing the configuration file was added

# The New Main Program

- The main program is changed significantly to meet the new requirements
  - The program should be able to support multiple experiments requested by the user
    - **Existing Program:** Returns the calculated solution by executing the user-selected algorithm once (Note: Numerous iterative calculations are performed during this single execution)
    - **New Requirement:** The program will be modified to execute the user-selected algorithm  $N$  times (numExp) and return the **best solution** (bestSolution) among them!
  - Experimental settings and other information should be read from a setup file provided by the user
    - Existing Program: Receives 3 pieces of information from the user via a terminal-based user interface: (1) Problem Type (Numeric/TSP), (2) Solution Algorithm (SA, FC, GD), (3) Filename where the problem is defined
    - New Requirement: To receive the 3 pieces of information above and various parameter values (delta, limitStuck, dx, numRestart, limitEval, numExp) from the user, the program will receive only the name of the setup file (where all information is pre-entered) and process the contents of the setup file to set the various parameters!

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

# Adding More Algorithms and Classes

- Modified User Interface and setup file:
  - Modified User Interface: The user only enters the setup file name in the terminal
  - Text File Example (Note: Lines starting with # are ignored):

```
2 # Select the problem type:  
3 #   1. Numerical Optimization  
4 ∵#   2. TSP  
5 | Enter the number (pType) : 2  
6 #  
7 #   Enter the name of the file : problem/Convex.txt  
8 #   Enter the name of the file : problem/Griewank.txt  
9 #   Enter the name of the file : problem/Ackley.txt  
10 #  Enter the name of the file : problem/tsp30.txt  
11 ∵#  Enter the name of the file : problem/tsp50.txt  
12 | Enter the name of the file : problem/tsp100.txt  
13 #  
14 # Select the search algorithm:  
15 # Hill Climbing algorithms:  
16 #   1. Steepest-Ascent  
17 #   2. First-Choice  
18 #   3. Stochastic  
19 #   4. Gradient Descent  
20 # Metaheuristic algorithms:  
21 ∵#   5. Simulated Annealing  
22 | Enter the number (aType) : 2  
23 #
```

```
24 # If you are solving a function optimization problem,  
25 ∵# what should be the step size for axis-parallel mutation?  
26 | Mutation step size (delta) : 0.01  
27 #  
28 # If your algorithm choice is 2 or 3,  
29 ∵# what should be the number of consecutive iterations without improvement?  
30 | Give the number of iterations (limitStuck) : 1000  
31 #  
32 # If your algorithm choice is 4 (gradient descent),  
33 ∵# what should be the update step size and increment for derivative?  
34 | Update rate for gradient descent (alpha) : 0.01  
35 | Increment for calculating derivative (dx) : 10 ** (-4)  
36 #  
37 # If you want a random-restart hill climbing,  
38 # enter the number of restart.  
39 ∵# Enter 1 if you do not want a random-restart.  
40 | Number of restarts (numRestart) : 10  
41 #  
42 # If you are running a metaheuristic algorithm,  
43 ∵# what should be the total number of evaluations until termination?  
44 | Enter the number (limitEval) : 100000  
45 #  
46 ∵# Enter the total number of experiments  
47 | Enter the number (numExp) : 10
```

# The New Main Program

- `main()`:
  - (`readPlanAndCreate`) Reads information from a setup file and
    - creates a problem `p` (`Problem` object) to be solved, and
    - creates an optimizer `alg` (`Optimizer` object) to be used
  - Conducts experiments and obtains the result (`conductExperiment`)
  - Describes the problem just solved (`p.describe`)
  - Shows the settings of experiment (`alg.displayNumExp` and `alg.displaySetting`)
  - Reports the result of experiment (`p.report`)

```
def main():
    p, alg = readPlanAndCreate()      # Setup and create (problem, algorithm)
    conductExperiment(p, alg)        # Conduct experiment & produce results
    p.describe()                    # Describe the problem solved
    alg.displayNumExp()             # Total number of experiments
    alg.displaySetting()            # Show the algorithm settings
    p.report()                      # Report result
```

# The New Main Program

- **readPlanAndCreate():**
  - Reads setup information from a file and stores them in the dictionary **parameters** (**readValidPlan**)
  - Creates **Problem** object and store it in **p** (**createProblem**)
  - Creates **Optimizer** object and store it in **alg** (**createOptimizer**)
  - Returns **p** and **alg**

```
def readPlanAndCreate():
    parameters = readValidPlan()  # Read and store in 'parameters'
    p = createProblem(parameters)
    alg = createOptimizer(parameters)
    return p, alg
```

# The New Main Program

- **readValidPlan():**
  - Reads setup information from a file and stores them in the dictionary **parameters** (**readPlan**)
  - Keeps querying the user if gradient descent is chosen for TSP
    - If the Gradient Descent solution is selected for the TSP problem (Error!), repeatedly call the **readPlan** function to receive a different setup file.
      - ✓ TSP probem : parameters['pType'] = 2
      - ✓ Grad Desc. algorithm : parameters['aType'] = 4
  - Returns **parameters**

```
def readValidPlan(): # Gradient Descent cannot solve TSP
    while True:
        parameters = readPlan()
        if parameters['pType'] == 2 and parameters['aType'] == 4:
            print("You cannot choose Gradient Descent")
            print(" unless you want a numerical optimization.")
        else:
            break
    return parameters
```

# The New Main Program

- **readPlan()**:

- Obtains a file name from the user # setup 파일의 이름을 입력 받기
  - Prepares a dictionary variable **parameters** to store the information

```
parameters = { 'pType':0, 'pFileName':'', 'aType':0, 'delta':0,  
               'limitStuck':0, 'alpha':0, 'dx':0, 'numRestart':0,  
               'limitEval':0, 'numExp':0 } # 딕셔너리 초기화
```

- Fills out **parameters** dictionary by reading the given setup file
    - All the information are numeric values except the name of the file containing specifics of the target problem (ex: problem/Convex.txt)
    - When reading the file line-by-line (**lineAfterComments**), lines beginning with '#' are all skipped
  - Returns **parameters** dictionary

# The New Main Program

- **readPlan():**

```
def readPlan():
    fileName = input("Enter the file name of experimental setting: ")
    infile = open(fileName, 'r')
    parameters = { 'pType':0, 'pFileName':'', 'aType':0, 'delta':0,
                    'limitStuck':0, 'alpha':0, 'dx':0, 'numRestart':0,
                    'limitEval':0, 'numExp':0 }
    parNames = list(parameters.keys())
    for i in range(len(parNames)):
        line = lineAfterComments(infile)
        if parNames[i] == 'pFileName':
            parameters[parNames[i]] = line.rstrip().split(':')[1]
        else:
            parameters[parNames[i]] = eval(line.rstrip().split(':')[1])
    infile.close()
    return parameters # Return a dictionary of parameters
```

# The New Main Program

- **lineAfterComments () :**
  - Skips the lines beginning with the symbol '#'
  - Returns the line that doesn't begin with '#'

```
def lineAfterComments(infile):      # Ignore lines beginning with '#'
    line = infile.readline()        # and then return the first line
    while line[0] == '#':           # with no '#'
        line = infile.readline()
    return line
```

- **createProblem(parameters) :**
  - Creates a **Numeric** or **Tsp** object and store in **p** depending on the type of problem chosen (i.e., **parameters['pType']**)
  - Sets some relevant variables of **p** in the class hierarchy according to the values in **parameters (p.setVariables)**
  - Returns a specific problem instance **p**

# The New Main Program

- **createOptimizer(parameters):**
  - Prepares a dictionary **optimizers** of algorithm class names (**optimizers**) that can be indexed by **aType** (= `parameters['aType']`)

```
optimizers = { 1: 'SteepestAscent()',  
              2: 'FirstChoice()',  
              3: 'Stochastic()',  
              4: 'GradientDescent()',  
              5: 'SimulatedAnnealing()' } # 선택 가능한 알고리즘 목록
```
  - Creates an object **alg** of the targeted algorithm by applying the **eval** function to the string of the name of the algorithm class (`eval(optimizers[aType])`)
  - Sets the class variables of **alg** with the values in **parameters** (`alg.setVariables`)
  - Returns **alg** as the created optimizer object

# The New Main Program

- **conductExperiment(p, alg):**
  - Solves the problem **p** with the chosen optimizer **alg** and collects the result of each individual experiment
    - If the chosen algorithm is a hill climber, then the random restart algorithm is called (**alg.randomRestart**)
    - Otherwise, its **run** method is called (**alg.run**)

```
def conductExperiment(p, alg):
    aType = alg.getType()
    if 1 <= aType <= 4:
        # 모든 Hill-Climbing 기법은 randomRestart를 통해서 실행됨
        # randomRestart 내부에서 run 을 호출함
        alg.randomRestart(p)
    else:
        # 반면, Meta-Heuristics 기법은 run 함수를 직접 실행함
        alg.run(p)
```

- Repeats experiment multiple times (**numExp = parameters[numExp]**) if requested and collects the results
- Finally, stores the summary of the best result (**p.storeExpResult**)

# Changes to ‘Problem’ Class

- For recording the results of experiments, the **Problem** class is revised to have the following additional variables:
  - **pFileName**: name of the file containing problem specifics
  - The following 6 values are calculated in the `main.py` by `conductExperiment` and are stored in the `problem` instance by calling `p.storeExpResult`
  - **bestSolution**: The best solution found so far
    - best solution found in  $n$  different experiments ( $n = \text{numExp} = \text{parameters}[\text{numExp}]$ )
  - **bestMinimum**: objective value of **bestSolution**
  - **avgMinimum**: average objective value of the best solutions obtained from  $n$  experiments (`sumOfMinimum / average`)
    - **sumOfMinimum**: Calculated in `main.py`, where the objective value obtained during each iteration over `numExp` is called `newMinimum`, and `sumOfMinimum += newMinimum`
  - **avgNumEval**: average number of evaluations made in  $n$  different experiments
  - **sumOfNumEval**: total number of evaluations made all through  $n$  experiments
  - **avgWhen**:
    - average iteration when the best solution first appears in  $n$  experiments
    - A meaningful value only in `MetaHeuristics`-based algorithms
    - `sumOfWhen += alg.getWhenBestFound()`, and `avgWhen` is the average value of `sumOfWhen`

## Changes to ‘Problem’ Class

- Accordingly, several new methods are added to handle those variables
  - Three new accessors `getSolution`, `getValue`, `getNumEval` are necessary for `conductExperiment` of the main program to conduct multiple experiments
  - The new method `storeExpResult` is also necessary for `conductExperiment` to store the result of experiment after finishing all the experiments

```
results = (bestSolution, bestMinimum, avgMinimum,  
          avgNumEval, sumOfNumEval, avgWhen)  
p.storeExpResult(results)
```

Part of `conductExperiment`  
function in `main.py`

# Changes to ‘Problem’ Class

- The `report` method has been revised to display the summary result of multiple experiments in an organized fashion
  - `report` in the base class prints messages that are common to both numerical optimization problem and TSP
  - Those in the subclasses print messages specific to the type of problem just solved
  - Therefore, the `report` methods in the `Numeric` and `TSP` classes call the report method of the parent class (`Problem.report(self)`) and then print their respective specialized information.

```
def report(self):  
    avgMinimum = round(self._avgMinimum, 3)  
    print()  
    print("Average objective value: {0:,.3f}".format(avgMinimum))  
    Problem.report(self)  
    print("Best solution found:")  
    print(self.coordinate()) # Convert list to tuple  
    print("Best value: {0:,.3f}".format(self._bestMinimum))  
    self.reportNumEvals()
```

report method from Numeric

```
def report(self):  
    avgMinimum = round(self._avgMinimum)  
    print()  
    print("Average tour cost: {0:,.3f}".format(avgMinimum))  
    Problem.report(self)  
    print("Best tour found:")  
    self.tenPerRow() # Print 10 cities per row  
    print("Best tour cost: {0:,.3f}" \  
          .format(round(self._bestMinimum)))  
    self.reportNumEvals()
```

report method from TSP

Refer to the next page

# Changes to ‘Problem’ Class

- The `reportNumEvals` method prints out the total number of evaluations regardless of the problem type

- However, it does nothing when the algorithm used is MetaHeuristics (simulated annealing or GA) because the number of evaluations for them is predetermined

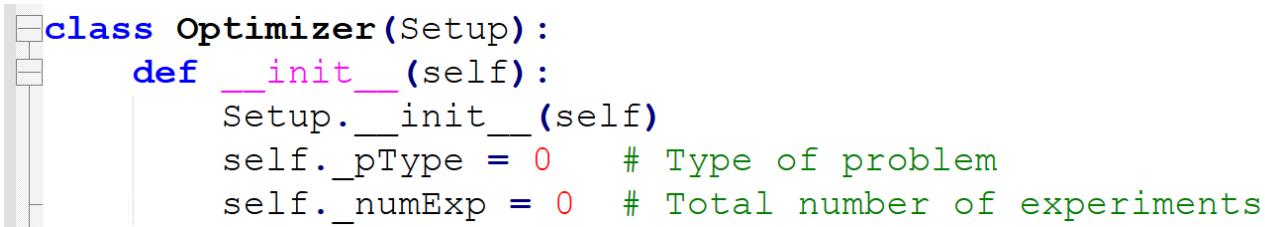
```
def reportNumEvals(self):
    if 1 <= self._aType <= 4:
        print()
        print("Total number of evaluations: {0:,}"
              .format(self._sumOfNumEval))
```

1. Steepest-Ascent
2. First-Choice
3. Stochastic
4. Gradient Descent

- It is separated from `report` because we want the result messages printed out in some appropriate order when they are mixed together with the messages generated from the subclasses
  - Call to `reportNumEvals` is made within `report` of the subclasses at its last line (Referring to the previous page)

# ‘Optimizer’ Class

- **Optimizer** has two variables `pType` and `numExp` to store common information about experimental settings:
  - The methods in the class hierarchy of `Optimizer` are extended versions of those previously existed in `HillClimbing`
  - New accessor methods `getWhenBestFound` (in `MetaHeuristics`) and `getNumExp` (in `Optimizer`) are added for being used by the `conductExperiment` function in the main program

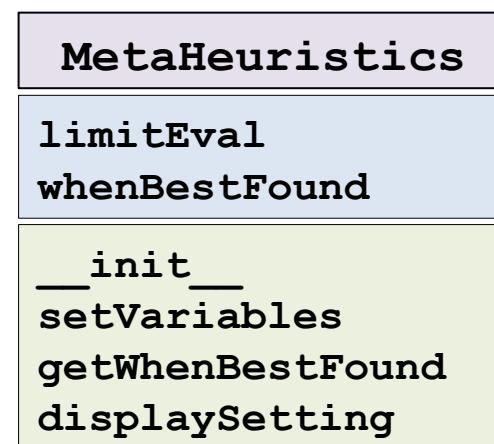
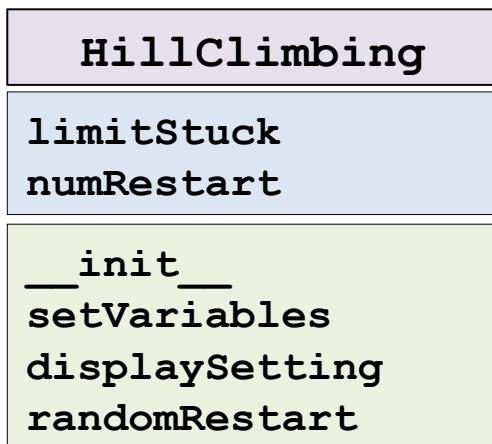


```
class Optimizer(Setup):
    def __init__(self):
        Setup.__init__(self)
        self._pType = 0      # Type of problem
        self._numExp = 0     # Total number of experiments
```

- Both ‘random.py’ and ‘math.py’ should be imported to the ‘optimizer.py’ file
  - the methods for stochastic hill climbing and simulated annealing algorithms need them

# 'Optimizer' Class

- **HillClimbing** now has only two variables: `limitStuck` and `numRestart`
  - `pType` has moved up to `optimizer`
- **MetaHeuristics** is a parent of **SimulatedAnnealing** and **GA**
  - **MetaHeuristics** has two variables: `limitEval` and `whenBestFound`
  - `displaySettings` method prints out `limitEval` (total number of evaluations until termination)



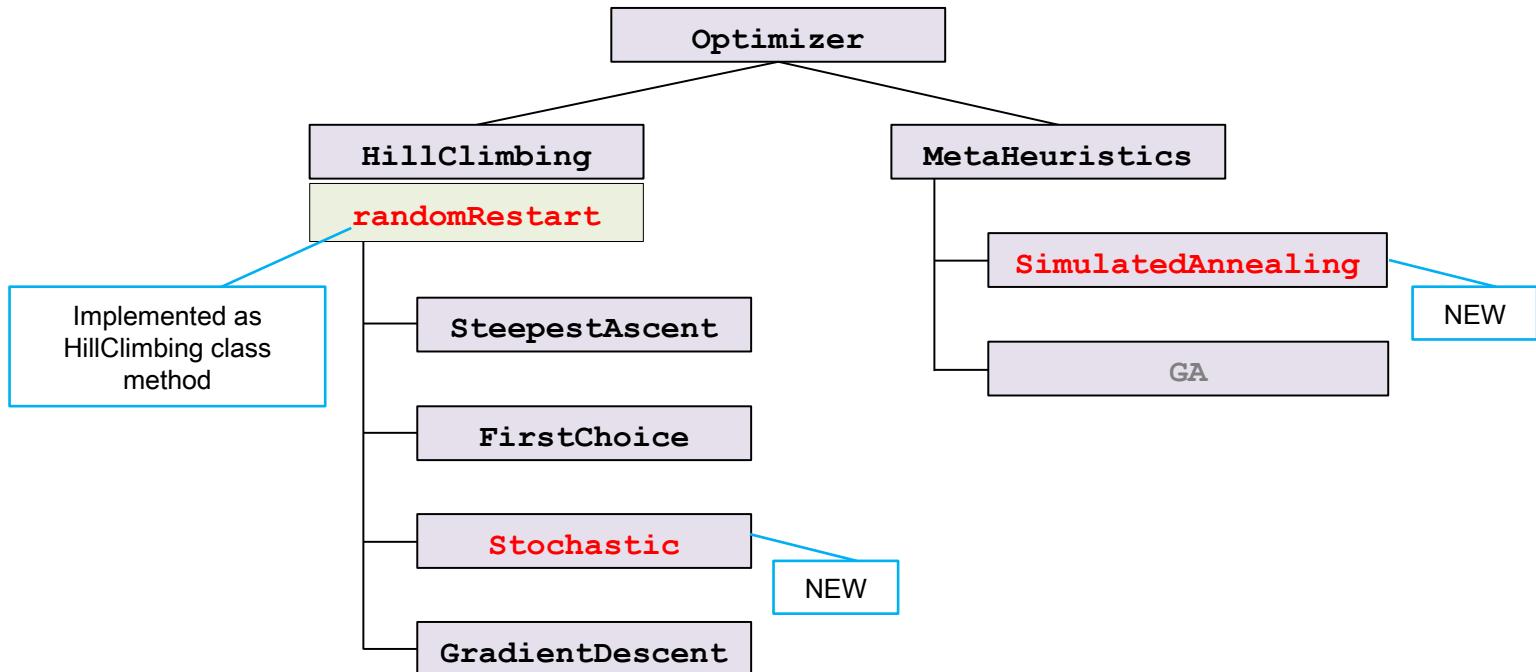
# Changes to ‘HillClimbing’ Class

- Changes of variables:
  - Under the new `Optimizer` class hierarchy, the variable `pType` is moved up from `HillClimbing` to `Optimizer`
  - A variable `numRestart` is newly added (the `setVariables` method is revised accordingly)
- Changes to the `displaySetting` method:
  - The printing of mutation step size is moved up to `Optimizer`
  - Now it prints out setting information related only to the variables of its own class `HillClimbing`

```
def displaySetting(self):  
    if self._numRestart > 1:  
        print("Number of random restarts:", self._numRestart)  
        print()  
    Optimizer.displaySetting(self)  
    if 2 <= self._aType <= 3: # First-Choice, Stochastic  
        print("Max evaluations with no improvement: {0:,} iterations"  
              .format(self._limitStuck))
```

# Adding three search algorithm

- (TO-DO) Add 3 new Search Algorithms to the existing code :
  - HillClimbing > Stochastic(stochastic hill climbing) and randomRestart, then MetaHeuristics > SimulatedAnnealing
  - Note that `randomRestart` is not implemented as a separate class but as a method within the `HillClimbing` class.



# Changes to 'HillClimbing' Class for randomRestart

- A new method `randomRestart` is added to HillClimbing class
  - It keeps calling `self.run` for a given number of times (`self._numRestart`), and stores the best solution found (`p.storeResult(bestSolution, bestMinimum)`)
  - HillClimbing overall call sequence:
    - Note: `MetaHeuristics` do not use the `randomRestart` method

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

```
if 1 <= aType <= 4:  
    alg.randomRestart(p)  
else:  
    alg.run(p)  
  
from conduct Experiment  
function in main.py..
```

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

**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)
```

`def run(self, p):`

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

# Stochastic Hill Climbing (Stochastic Class)

- The algorithm of stochastic hill climbing is the same as first-choice hill climbing except the way a successor is chosen
  - First-choice hill climbing generates a single successor randomly
  - Stochastic hill climbing generates multiple neighbors and then selects one from them at random by a probability proportional to the solution quality
  - The algorithm is implemented as the `run` method as before

## <SteepestDescent run method>

- Generate  $2N$  neighbor solutions and select the single best solution using the `bestOf` function,
- Compare with `current solution`
- Choose if better; otherwise terminate

## <First-Choice run method>

- Randomly generate 1 neighbor solution,
- compare with current solution
  - Choose if better,
  - Otherwise, terminate if `self._limitStuck` consecutive bad solutions are encountered



## <Stochastic run method>

- Generate  $2N$  neighbor solutions
- Select one randomly using the `stochasticBest`,
- compare with current solution
  - Choose if better,
  - Otherwise, terminate if `self._limitStuck` consecutive bad solutions are encountered

CAUTION!! The actual implementation proceeds this way (like `First-Choice`, by selecting only solutions better than the current one). This is because, in this lecture, `HillClimbing` is assumed to be a category that includes techniques that only advance toward a better side...

# Stochastic Hill Climbing (Stochastic Class)

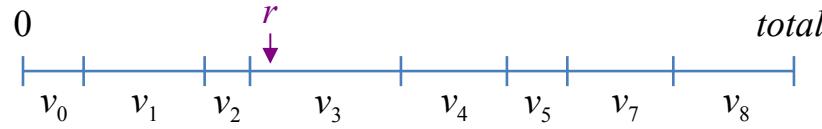
- **stochasticBest(self, neighbors, p):**
  - A function that probabilistically selects one solution from **2N neighbor** solutions
  - It does not select completely randomly, but assigns a probability proportional to the **solution quality**
  - Ex: Situation during the  $k$ -th iteration in a problem that aims to **minimize** a given expression
    - current soln :  $x = \dots$
    - Two neighbor solutions,  $y$  and  $z$ , were generated, and their *obj value* are as follows:
      - ✓  $f(y) = 15$
      - ✓  $f(z) = 5$
    - Since it is a **minimize** problem, solution  $z$  is better  $\Rightarrow$  Assign a higher selection **probability** to the better solution
  - ✓ Probability of selecting solution  $y$  :  $\frac{\frac{1}{15}}{\frac{1}{15} + \frac{1}{5}} = \frac{\frac{1}{15}}{\frac{4}{15}} = \frac{1}{4} = 25\%$
  - ✓ Probability of selecting solution  $z$  :  $\frac{\frac{1}{5}}{\frac{1}{15} + \frac{1}{5}} = \frac{\frac{1}{5}}{\frac{4}{15}} = \frac{3}{4} = 75\%$

# Stochastic Hill Climbing (Stochastic Class)

- **stochasticBest(self, neighbors, p):**
  - Probabilistic selection: How to select  $y$  with 0.25 probability and  $z$  with 0.75 probability??
  - [1] Use a probability value between 0 and 1 to search sequentially
    - Generate a **random value** between [0, 1] and store it in  $r$
    - If the value of  $r$  is less than (or equal to) 0.25, select  $y$ ; otherwise, select  $z$
  - [2] Use the **numpy** library
    - `my_choice = np.random.choice(a=[x,y], p=[0.25,0.75])`
    - Note : [numpy\\_random documentation](#)
  - however, ...
    - If  $f(y)=1$  and  $f(z)=-1$ , the denominator becomes 0 (divide-by-zero.)
    - Depending on the situation, a negative probability might result.
    - What is the implementation method that **considers general situations?** (*Next page*)

# Stochastic Hill Climbing (Stochastic Class)

- **stochasticBest(self, neighbors, p):**
  - Implementation method considering the general situation:
  - Obtains a list of evaluation values of **neighbors** and store in **valuesForMin**
    - (Original problem) **Smaller values are better**
    - However, to make the probability-based selection calculation easier, the following operation is performed
  - Converts the list to the one in which larger values are better (**valuesForMax**)
    - Changes the values so that the larger the value, the higher the selection probability
    - Each original value is subtracted from a large enough value (ex.  $\max(valuesForMin)+1$  to avoid zero)
    - The converted values are stored in the **valuesForMax** list
  - Chooses at random from **valuesForMax** with probability proportional to its value



- Returns the chosen neighbor together with its original evaluation value

예) 9 solutions were made, and  $v_0$  to  $v_8$  are the **valuesForMax**. A value is drawn randomly from the list. After generating a random number  $r$  within  $[0, \sum(valuesForMax)]$ , one solution is selected along with its index, like in the example

# Simulated Annealing (SimulatedAnnealing Class)

## • Working Principle

- Generates one **next solution** randomly (just like **first-choice**)
- Selects the **next solution** if it is **better** than the current solution.
- Selects the **next solution probabilistically** even if it is **worse** than the current solution.
  - Uses the probability  $\exp(-dE/T)$  for selection

- 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*

Calculate the current state from the initial state

$\Delta E \leftarrow next.VALUE - current.VALUE$

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

if  $\Delta E < 0$  then *current* ← *next*

Assumes a minimization problem → Lower value is better

else *current* ← *next* only with probability  $e^{-\Delta E/T}$

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

- **Additional Member Variable:** There is one variable **numSample** storing the number of samples used to heuristically determine an **initial temperature** ( 초기 온도값 )
  - It is currently preset to 100 (in **SimulatedAnnealing Class**)
  - Used in **initTemp** function (See *next slide*)
  - Used to generate the initial temperature value.

# Simulated Annealing (SimulatedAnnealing Class)

- **initTemp(self, p):**
  - Working method
    - Calculates the temperature  $t$  such that  $\exp(-dE/t) = 0.5$ , and returns  $t$
    - That is, it finds the temperature  $t$  where the probability of selecting a bad solution is 0.5 and returns it.
    - **Why?** It is reasonable to use a  $t$  that satisfies the above condition as the initial temperature.
  - Implementation
    - Takes  $k$  ( $= \text{self._numSample}$ ) random samples and their neighbors from the domain of problem  $p$
    - The process of generating an arbitrary initial value, randomly generating its neighboring solution, recording the difference, and calculating the average difference value is repeated  $k$  times.
    - Calculates the average difference between two solutions ( $= dE$  average value)
    - Sets the initial  $t$  value using the calculated average value of  $dE$  such that the probability of selecting a bad solution is 0.5 initially.

# Simulated Annealing (SimulatedAnnealing Class)

- **initTemp(self, p):**
  - **Working method**
    - Calculates the temperature  $t$  such that  $\exp(-dE/t) = 0.5$ , and returns  $t$
    - That is, it finds the temperature  $t$  where the probability of selecting a **bad solution is 0.5** and returns it.
  - **Implementation**
    - Implemented as following:
      - ✓ For  $m$ -th iteration in `range(self._numSample):`
        - » generate a random initial solution and evaluate ( $= v_0$ )
        - » generate a random mutation and evaluate ( $= v_1$ )
        - » `calc_diffs(m) = abs(v1-v0)`
      - ✓ Calculates the average  $dE$  of their differences
        - » `dE = sum( diff ) / self._numSample`
      - ✓ `return t = dE / math.log(2) # Initial Temperature Setting. Why is it this way? (Next Page)`

# Simulated Annealing (SimulatedAnnealing Class)

- ~ **initTemp(self, p):**
  - Find the temperature **T** at which the probability of selecting a **bad solution** becomes **0.5**.
  - Probability of selecting a bad **soln** (solution) :  $p = e^{-dE/T}$
  - Find **T** where **p=0.5**
    - $p = \frac{1}{2} = e^{-dE/T}$
    - $\log\left(\frac{1}{2}\right) = \log e^{-dE/T} = -\frac{dE}{T}$
    - $-\log(2) = -\frac{dE}{T}$
    - $T = \frac{dE}{\log(2)}$
    - $dE$  = The difference in value calculated probabilistically

# Simulated Annealing (SimulatedAnnealing Class)

- ~ **initTemp(self, p):**
  - **Summary of Operation:** Find the initial temperature  $t$  at which the probability of selecting a bad solution becomes 0.5.
    - Repeat for numSample times
      - ✓ Generate a random solution  $c_0$  and one random mutant (next solution)  $c_1$  from it.
      - ✓ Calculate the obj value  $v_0$  for  $c_0$  and the obj value  $v_1$  for  $c_1$ .
    - $dE = \text{average of } \text{abs}(v_1 - v_0)$ 
      - ✓ That is,  $dE$  calculates the average difference between **two objective values**.
      - ✓  $t = dE / \text{math.log}(2)$  you can find the initial temperature  $t$  where the probability of selecting a bad solution becomes **0.5**.

# Simulated Annealing (SimulatedAnnealing Class)

- **tschedule(self, t):**

- Calculates the next temperature using a simple formula, and returns it

```
def tschedule(self, t):
    return t * (1 - (1 / 10**4))
```

- The temperature schedule values can be used in the form of a pre-calculated **list (array)**, or it can be implemented, as shown above, to take the **current temperature** as input and calculate the **next temperature**.

# Simulated Annealing (SimulatedAnnealing Class)

- `run()` method explanation:
  - Starts from a random initial point
    - The randomly selected initial solution is recorded as the **best solution**, and this record is **updated** whenever a better solution is found.
  - The solution is **iteratively updated**, and the process terminates when the temperature (`t`, `temperature`) reaches **0** or the number of iterations reaches `self._limitEval`.
    - it uses another variable `whenBestFound` to record when the best-so-far solution has first been found (it records the iteration counter value at that time)
  - The temperature decreases every iteration according to an annealing schedule (`self.tSchedule(t)`)
    - Note: As temperature gets lower the probability of bad solution get selected decreases.
  - The initial temperature is heuristically determined so that the probability of accepting a worse neighbor becomes 0.5 initially (`self.initTemp(p)`)
  - Generate a random neighboring solution based on the **current solution**,
    - If the neighbor is a **better solution**, it is **always selected**. If not:
      - ✓ The probability of accepting a worse neighbor is `exp(-dE/t)`,
      - ✓ `dE` is the difference of the evaluation values (= `valueNext - valueCurrent`)
      - ✓ `t` is the current temperature

# Search Algorithms: Object-Oriented Implementation (Part F)

# Contents

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

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.

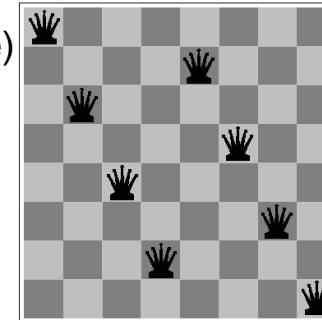
# Genetic Algorithm

- 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.”

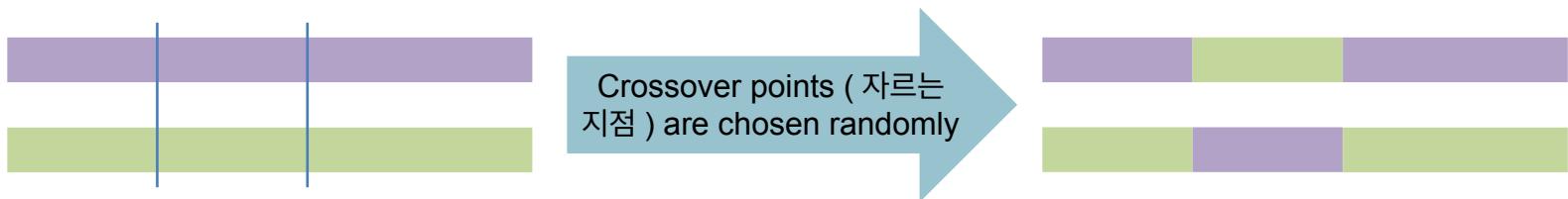
Simulated annealing and GA use meta-level heuristic → **metaheuristic algorithms**

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 ( 짹짓기 ) 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 ( 짹짓기 ) 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.

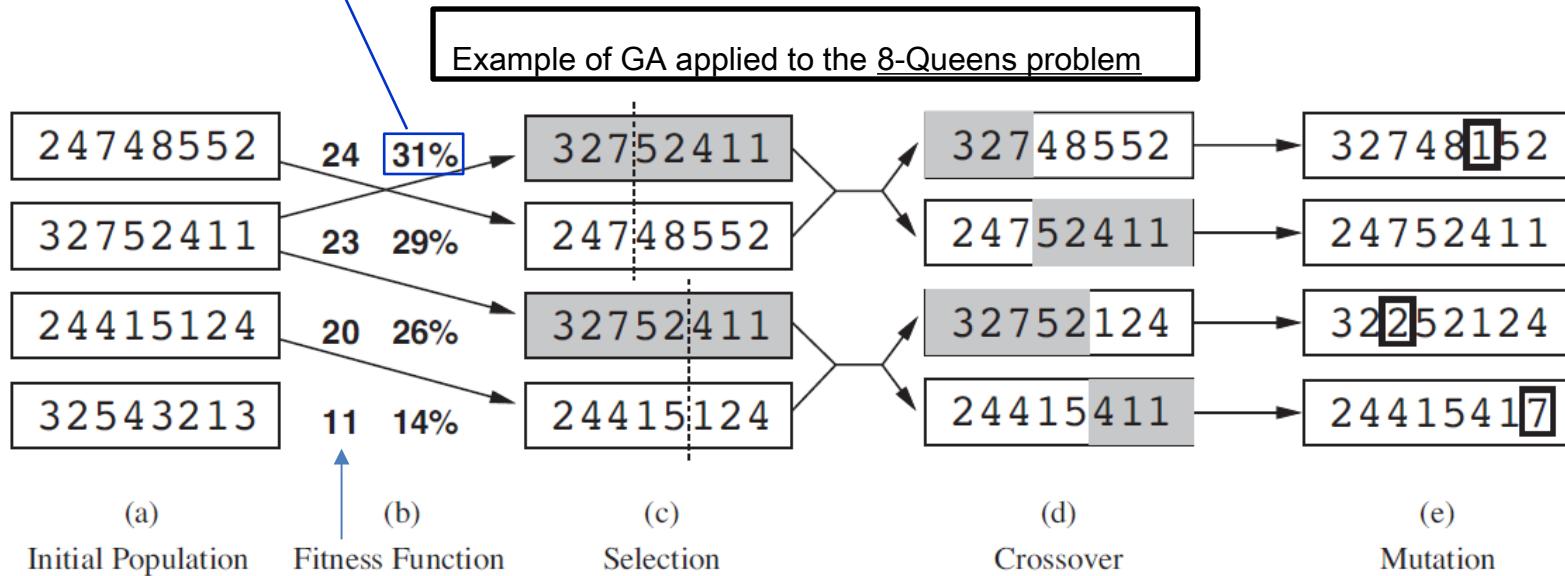
The choice of the crossover rate can impact the balance between exploration and exploitation in the genetic algorithm. A higher crossover rate promotes more exploration by creating new offspring solutions through recombination. On the other hand, a lower crossover rate favors exploitation by preserving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

# 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.

$$31\% = 24 / (24+23+20+11)$$

# 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*  $\leftarrow$  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."

$(x, y) \leftarrow$  SELECT-PARENTS(*population*, FITNESS-FN)      Select two solutions by fitness scores

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

**if** (small random probability) **then** *child*  $\leftarrow$  MUTATE(*child*)      Randomly change the solution

add *child* to *new\_population*      Add the newly generated solution to *new\_population*.

*population*  $\leftarrow$  *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

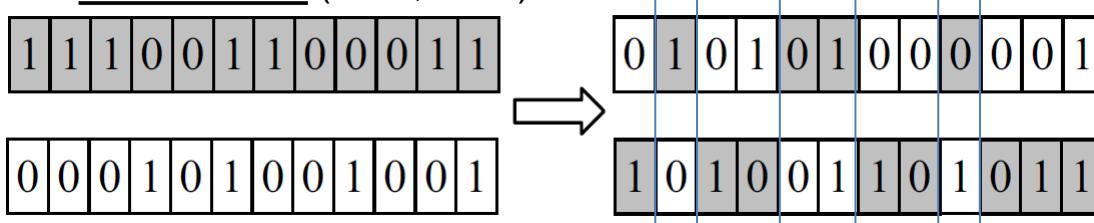
1. **Randomly select two (2) individuals with replacement** (i.e., the same solution can be selected twice) **without considering their fitness score**
2. Select the one with the best fitness as the winner (ties are broken randomly)

Tournament

- **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).

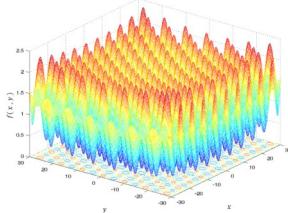
The crossover operation is performed to generate **two children** from **two parents**.

- 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

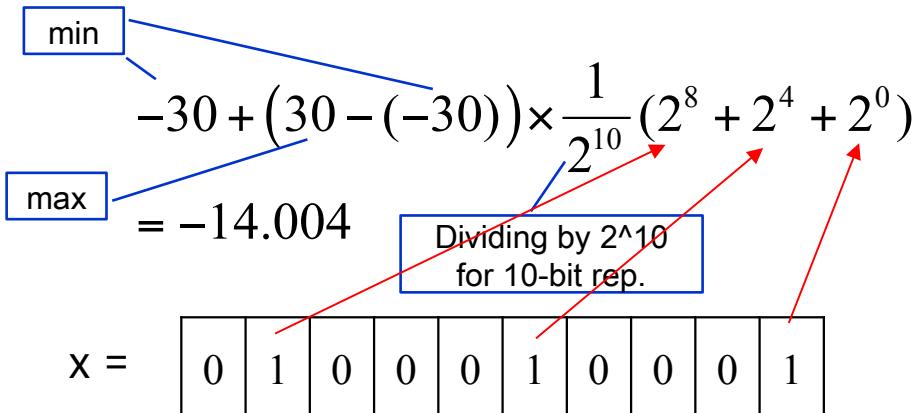


2-D Griewank function

Example: Binary encoding/decoding for numerical optimization

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

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



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.

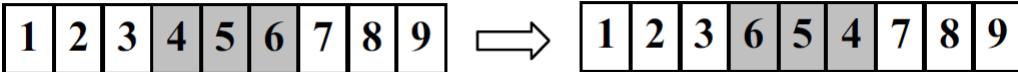
- In case of 10-bit, the maximum representation value is  $2^{10}-1$ .
- therefore, representing 10-bit by dividing by  $2^{10}$ , it becomes a real number between (0,1).
- This value is **scaled** and **shifted** to convert it into a number within the valid range.
- Note: as  $n$  in  $n$ -bit representation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

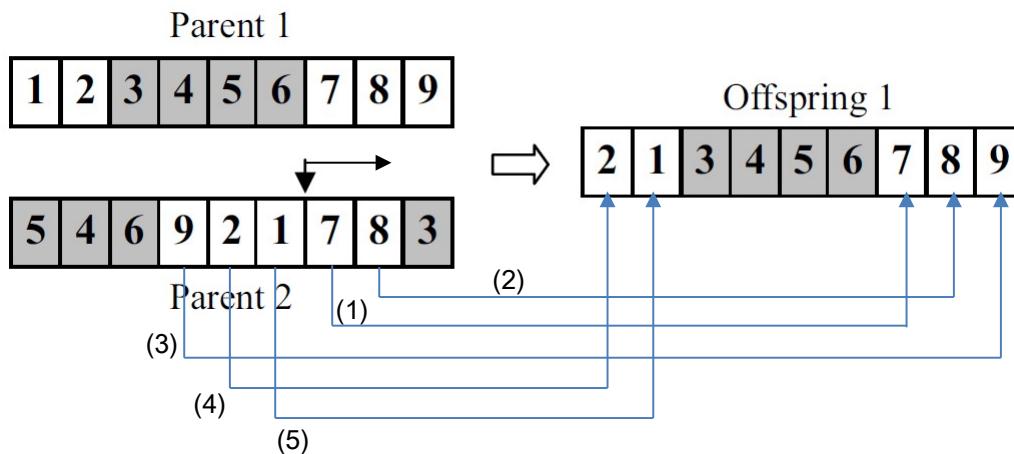
# Genetic Algorithm

- 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:
      - ✓  $[1,2,3,4] + [4,3,2,1] \Rightarrow [1,2,2,1]$
      - ✓ The original two solutions are valid, but the solution created by crossover is **invalid** (it **does not visit all cities**).

# Genetic Algorithm

- 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

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



- 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) IM- PLEMENTATION**

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

numerical  
opt.

TSP

| GA  |
|---|
| <b>popSize</b>  |
| <b>uXp</b>  |
| <b>mrF</b>  |
| <b>XR</b>   |
| <b>mR</b>   |
| <b>pC</b>   |
| <b>pM</b>   |
| <u>init</u><br><u>setVariables</u><br><u>displaySettings</u><br><u>run</u><br><b>evalAndFindBest</b><br><b>selectParents</b><br><b>selectTwo</b><br><b>binaryTournament</b> |

# 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.selectParents, p.crossover, p.mutation`), updating the population (= generate next solution set), and updating the best-so-far individual (`self.evalAndFindBest`)
  - 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 6<sup>th</sup> 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 probability `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):**
  - o Performs mutation to **ind** and returns it (Performs solution variation on a single solution)
  - o 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.
  - o 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):**
  - o 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)`
  - o In **Numeric**, the chromosome is decoded and then returned (**self.decode**)
  - o 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.binaryToDecimal`), 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
```

Fitness score

```
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
```

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

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

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

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

# Code sample for Numeric problems

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

```
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 - 1) * decimalValue / 2 ** r
```

**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

```
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

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

```
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

```
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**

Swap the numbers within the set range

- **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

# 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  $-30 \leq x_i \leq 30$  for  $1 \leq i \leq 5$

– Convex function:

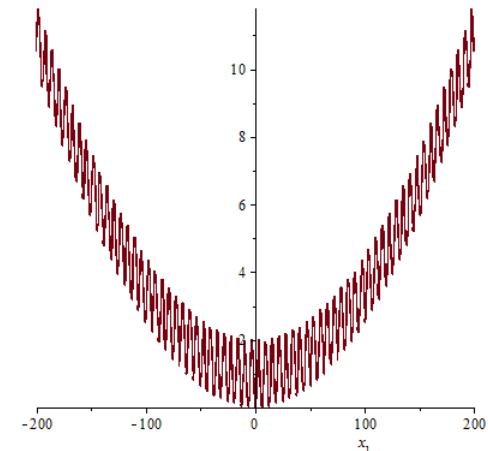
$$(x_1 - 2)^2 + 5(x_2 - 5)^2 + 8(x_3 + 8)^2 + 3(x_4 + 1)^2 + 6(x_5 - 7)^2$$

– 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

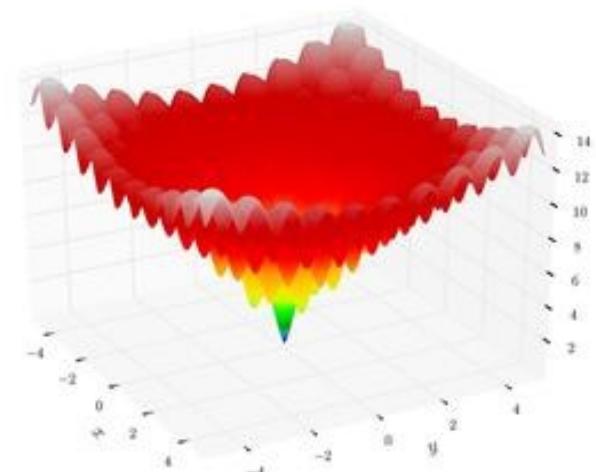
- Ackley function:

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

with a global minimum of 0 at all zeros

- We also try three versions of TSPs with different numbers of cities located in a  $100 \times 100$  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:  $10^{-4}$
- 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

$$x \leftarrow x - \alpha \nabla f(x)$$

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

# 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<br><b>0.0</b>        | 774,692            | 0.260<br>0.108               | 67,144             | 17.832<br><b>14.029</b>      | 12,182             |
| First Choice        | 0.0<br><b>0.0</b>        | 274,521            | 0.254<br><b>0.064</b>        | 38,824             | 18.214<br><b>14.108</b>      | 14,377             |
| Stochastic          | 0.0<br><b>0.0</b>        | 2,088,171          | 0.218<br><b>0.096</b>        | 387,008            | 18.879<br><b>16.729</b>      | 141,156            |
| Gradient Descent    | <b>0.0</b><br><b>0.0</b> | 199,935            | 0.216<br><b>0.118</b>        | 856,635            | 17.447<br><b>8.101</b>       | 5,234              |
| Simulated Annealing | 0.0<br><b>0.0</b>        | 500,000<br>63,565  | 0.367<br><b>0.145</b>        | 500,000<br>18,252  | 19.319<br><b>18.940</b>      | 500,000<br>5,541   |
| GA                  | 3.920<br><b>0.766</b>    | 500,000<br>227,140 | <b>0.036</b><br><b>0.015</b> | 500,000<br>220,390 | <b>0.214</b><br><b>0.141</b> | 500,000<br>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<br>525 | 6,846              | 782<br>678 | 20,211             | 1,223<br>1,145 | 88,254            |
| First Choice        | 424<br>408 | 30,154             | 578<br>561 | 57,207             | 903<br>869     | 131,450           |
| Stochastic          | 422<br>408 | 670,952            | 570<br>561 | 2,395,138          | 872<br>840     | 10,903,041        |
| Nearest Neighbor    | 509<br>455 |                    | 694<br>646 |                    | 958<br>918     |                   |
| Simulated Annealing | 412<br>408 | 500,000<br>28,831  | 577<br>559 | 500,000<br>43,962  | 829<br>804     | 500,000<br>69,084 |
| GA                  | 658<br>635 | 500,000<br>245,940 | 584<br>558 | 500,000<br>226,840 | 854<br>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)

# Linear Regression and $k$ -NN (Part A)

Theory

# Contents

- Understadning and Implementing Representative Machine Learning Techniques
  - Univariate Linear Regression
  - Multivariate Linear Regression
  - Nearest Neighbor Models
  - Implementation and Experiments

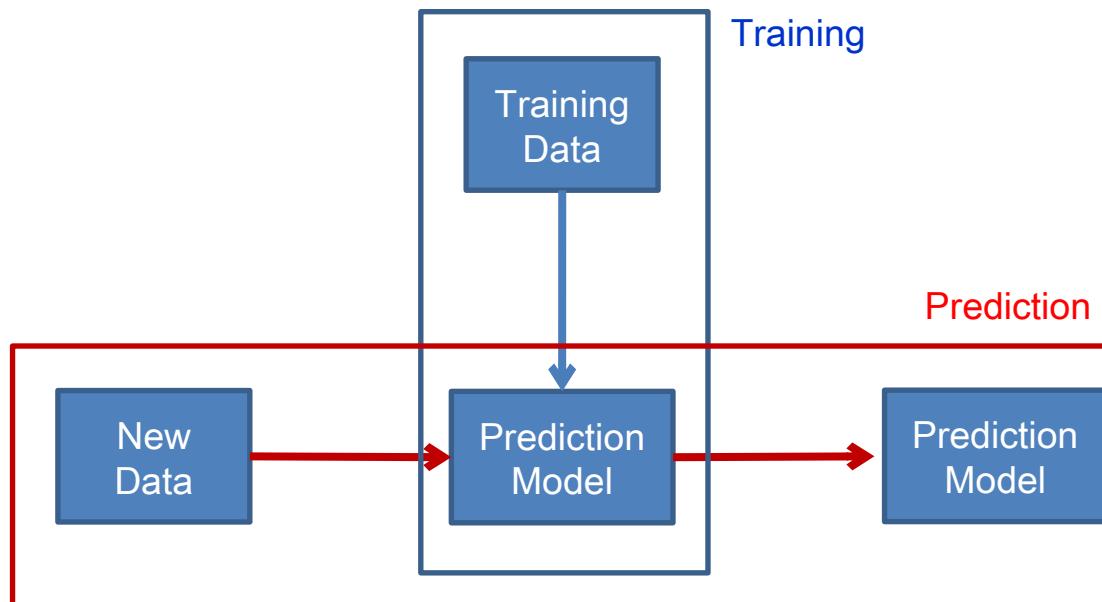
Next time...

In this lecture, we will study representative machine learning techniques: Linear Regression and Nearest Neighbor Model.



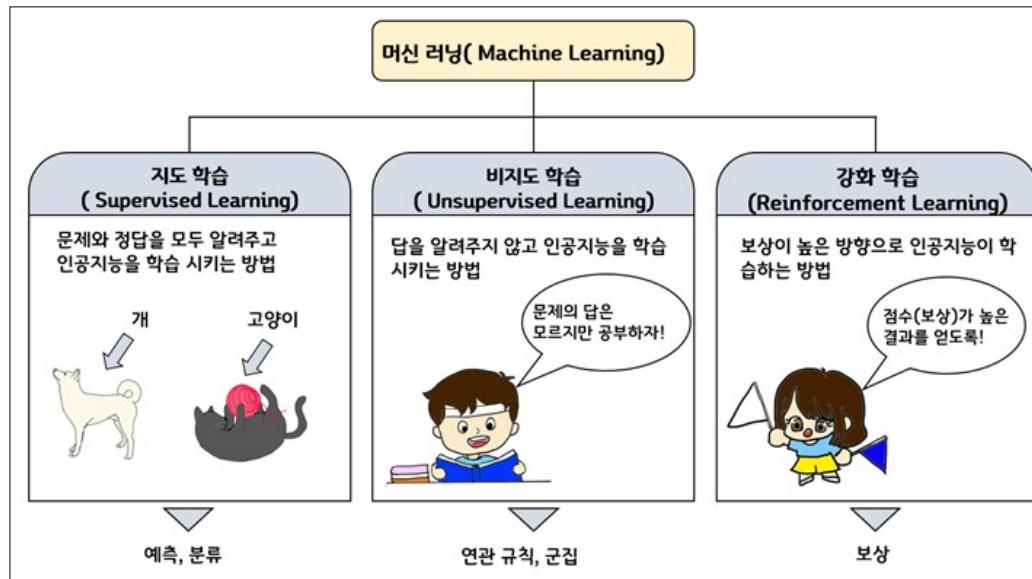
# INTRO

- Machine Learning ( 기계 학습 , 머신 러닝 )
  - A field of technology that develops algorithms and statistical models used by computer systems to perform pattern recognition and inference, and to perform tasks on their own without explicit instructions
  - Computer systems use machine learning algorithms to process large amounts of data and identify patterns.  
=> Through this, accurate results can be predicted.
  - In other words, Machine Learning is a technology that recognizes patterns autonomously through **learning** and performs accurate **prediction** through **inference**.



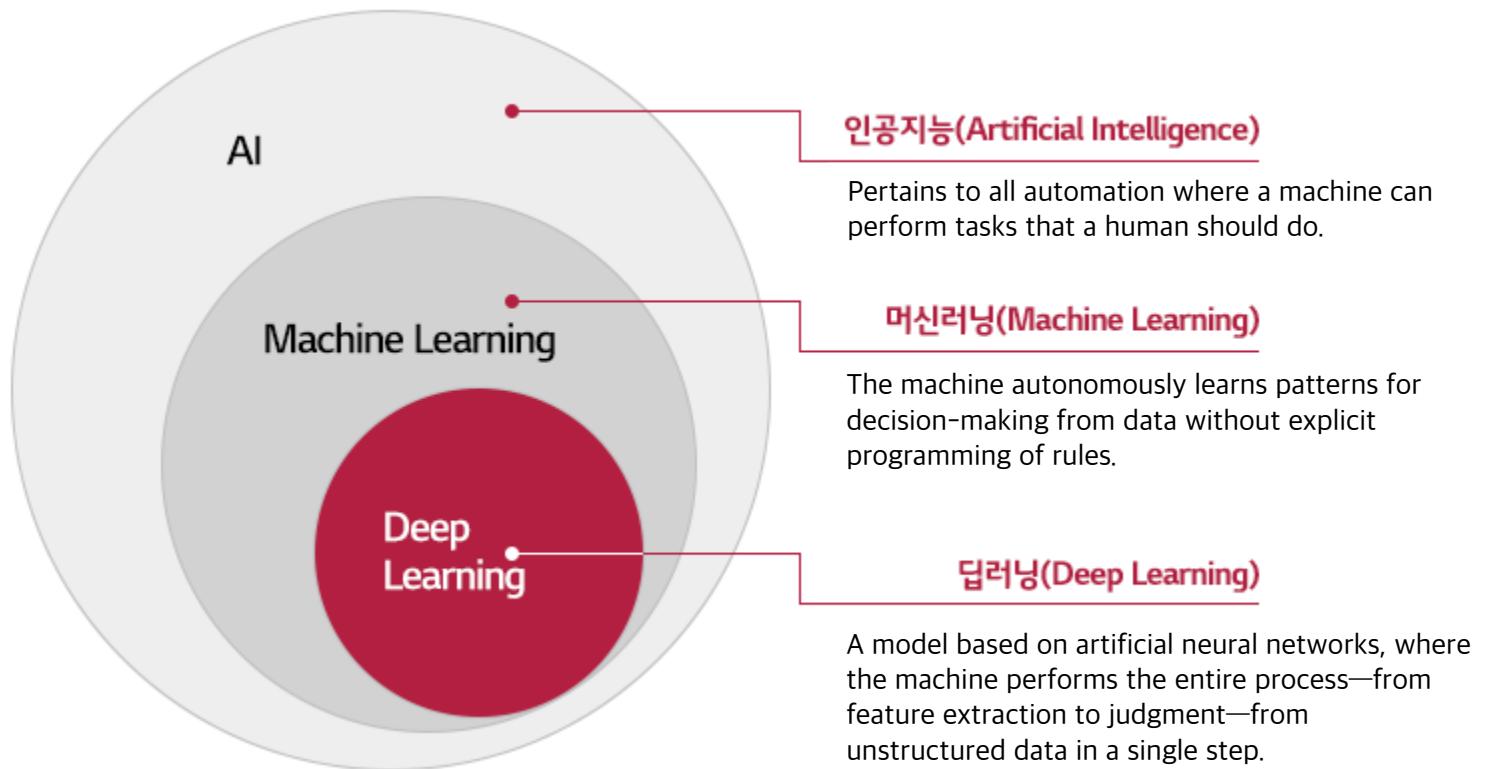
# INTRO

- Classification of Machine Learning
  - Machine learning is classified into **Supervised Learning**, **Unsupervised Learning**, and **Reinforcement Learning**.
    - Supervised Learning:** A method of training a machine to find patterns between data and answers using multiple (**data**, **answers**) as input, so that it can correctly predict the answer when new data is given.
    - Unsupervised Learning:** A method of training a machine to find patterns and correlations between data using multiple (**data**) as input.
    - Reinforcement Learning:** A method of training a machine using continuous (**actions**, **rewards**) as input, so that it autonomously finds the actions that lead to high rewards.



# INTRO

- AI( 인공지능 ), ML( 기계학습 ), DL( 딥러닝 ) Relationships



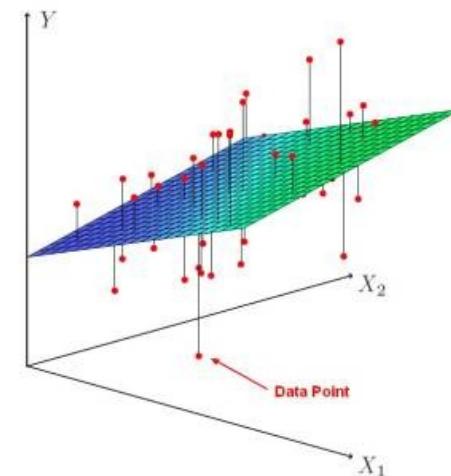
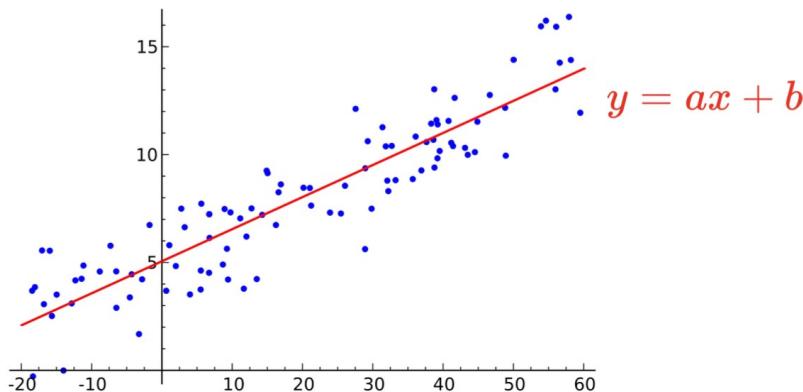
# **UNIVARIATE LINEAR RE- GRESSION**

# Regression

- The problem to solve
  - Input: Multiple pairs of  $(x_i, y_i)$ ,  $i = 1, 2, 3, 4, \dots, N$  ( $N$  = number of sample data)
    - $x_i$  : data (features)
    - $y_i$  : Answer/Label for the given data( $x_i$ )
    - ex: Room/Bathroom area (= feature) and monthly rent (= label)  
 $x_1 = 30 \text{ m}^2, y_1 = 30 \text{ 만원},$   
 $x_2 = 45 \text{ m}^2, y_2 = 40 \text{ 만원},$   
 $x_3 = 60 \text{ m}^2, y_3 = 60 \text{ 만원}, \dots$  In this situation with  $N=3$  sample data, given  
 $x_{\text{new}} = 52 \text{ m}^2$  what is the monthly rent  $y_{\text{new}} = ?$
  - Goal: To develop a machine learning model that understands the relationship between data and answers from multiple (**data**, **answer**) samples, and predicts the answer when new data is given!
- Approach: Use the Regression (회귀) technique
  - Regression (회귀)
    - A technique for modeling the correlation  $\mathbf{h}(\mathbf{x}) = \mathbf{y}$  between one or more **independent variables**  $\mathbf{x}$  and a single **dependent variable**  $\mathbf{y}$  (i.e., a method for calculating what the function  $\mathbf{h}$  should be).
    - A method to understand the influence of the **independent variable(s)** on the **dependent variable** and, based on this, to generate a model that predicts the value of the **dependent variable** corresponding to a given value of the **independent variable(s)**.

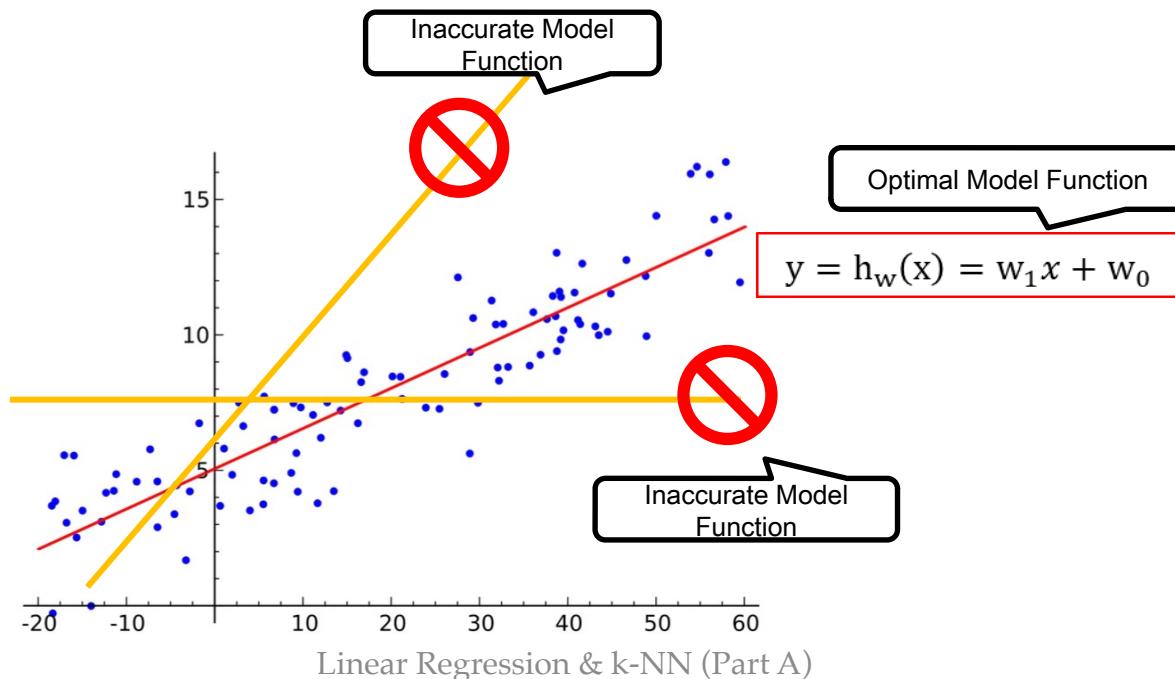
# Linear Regression

- Linear Regression (선형 회귀)
  - Regression
    - A technique for modeling the correlation  $h(x) = y$  between one or more **independent variables**  $x$  and a single **dependent variable**  $y$
  - Linear Regression
    - A method for finding the function  $h$  by assuming it is a **linear function**.
    - If the **independent variable** is a scalar,  $h$  is a **straight line graph**, and if the **independent variables** are multiple (a vector),  $h$  becomes a **plane in a multidimensional space**.
    - The case where the **independent variable** ( $x_i$ ) is a scalar is called the **Univariate Linear Regression** model,  
and the case where the **independent variable** ( $x_i$ ) is a vector composed of multiple values is called the **Multivariate Linear Regression** model.



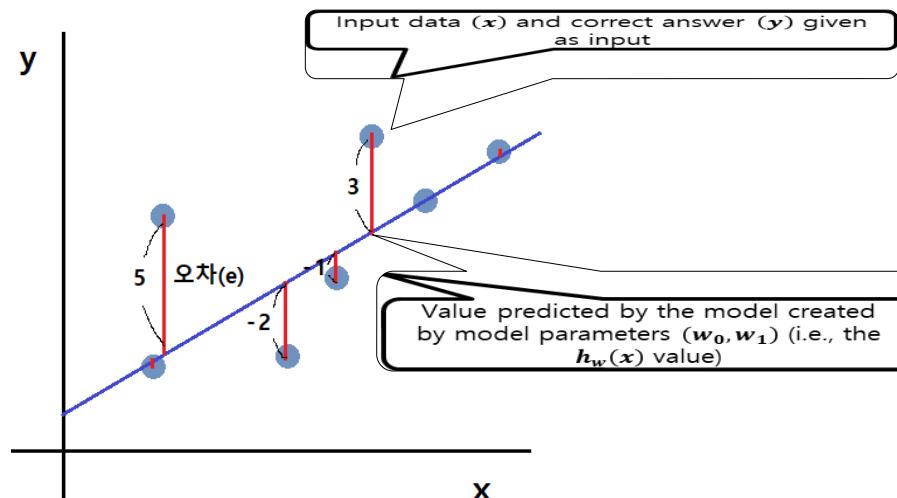
# Univariate Linear Regression

- ULR
  - A technique for linearly (=straight-line graph) modeling the correlation between a scalar **independent variable**  $x$  and a **dependent variable**  $y = h(x)$ .
  - Approach
    - Assume an arbitrary straight-line graph function  $h_w(x) = w_1x + w_0$  ( $w_1$ : slope/gradient,  $w_0$ :  $y$ -intercept).
    - Adjust the values of  $w_1$  and  $w_0$  to make the function  $h_w(x)$  represent/describe the entire dataset as accurately as possible.
    - But, what exactly does it mean to "most accurately" represent...?



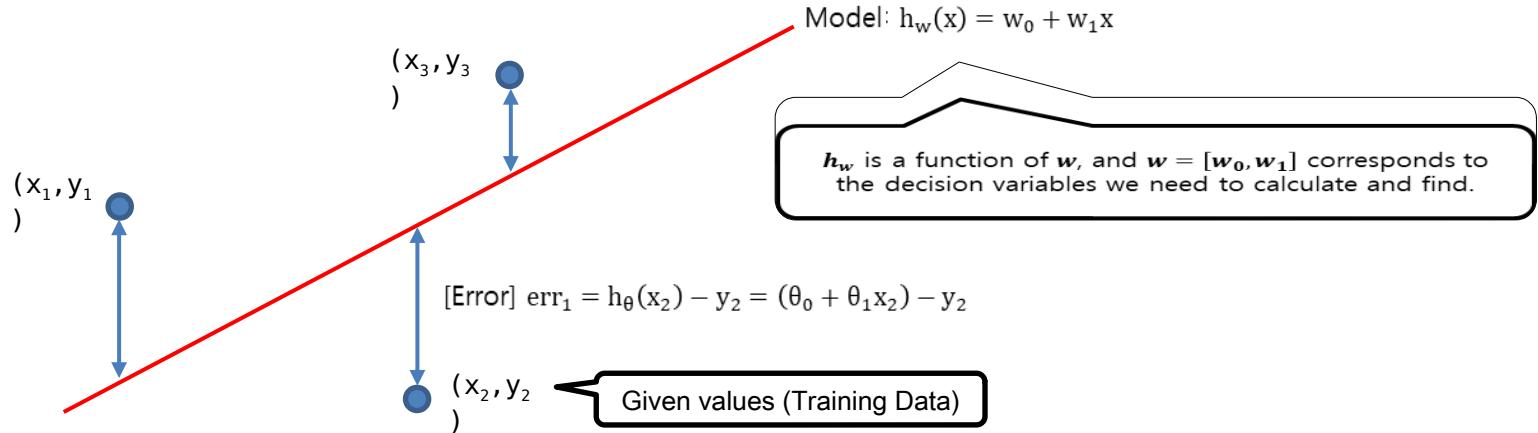
# Univariate Linear Regression

- ULR
  - Loss Function (오차함수)
    - The **core** of the ULR problem is to define a **Loss Function** to measure how accurately the function  $h_w(x)$  **represents** the entire dataset, and then to find the  $w_1, w_0$  values that **minimize the Loss Function**.
      - ✓ **Note:** For a given data pair  $(x_i, y_i)$  used for training, the returned value of  $h(x_i)$  is the **predicted value**, and  $y_i$  corresponds to the **correct answer** (ground truth).
      - ✓  $(x_i, y_i)$  are the values given for training, and to find the optimal function  $h_w(x)$ , we must find the optimal  $w_1, w_0$ .
    - Defining the Loss Function



# Univariate Linear Regression

- ULR
  - Defining the Loss Function
    - Error = Predicted Value – Actual Value
      - ✓ When  $(x, y)$  is given, the **Actual Value** (ground truth) is  $y$ , and the **Predicted Value** is  $h_w(x)$ .
      - ✓ The difference between the two is defined as the **Error**, and the **Optimal Model Function** can be found by searching for  $w_1, w_0$  that minimizes the sum of the errors over the entire training dataset.



- **(Modified) Error** =  $\sum(y_i - h_w(x_i))^2, \forall i = 1, 2, \dots, N$  // **For all given sample data...**
  - ✓ If the errors for the first and second training data are -5 and +5, respectively, the sum becomes 0 (=no error?), which is a misleading assessment. To prevent this erroneous judgment, the goal is to find  $w_0, w_1$  that minimizes the **sum of the squared errors** for all data, by calculating the **square of each error**.

# Univariate Linear Regression

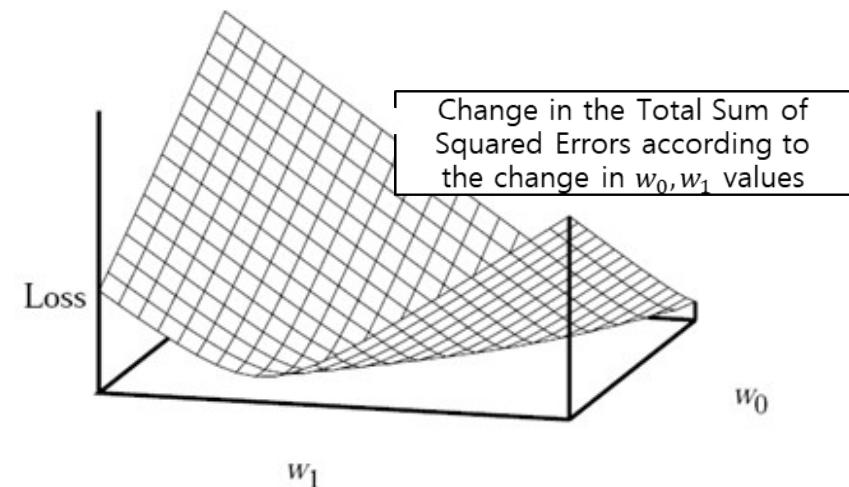
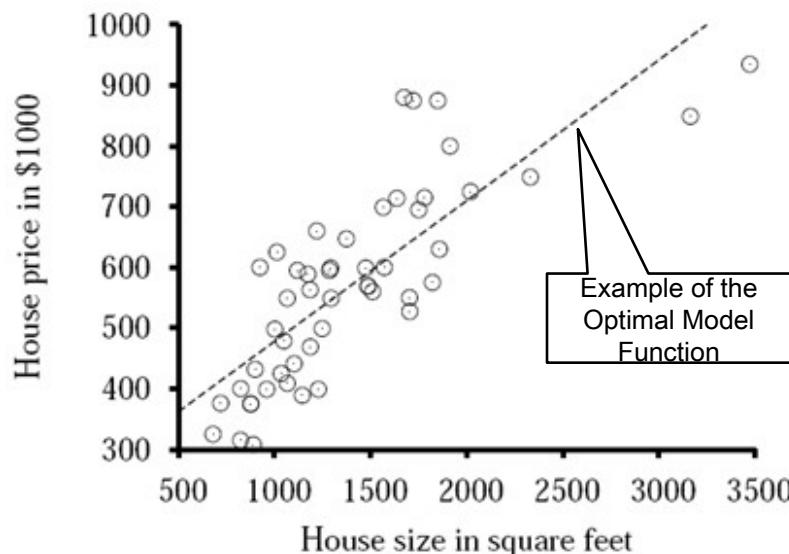
- The task of finding  $h_w$  that best fits the data

$$h_w(x) = w_1x + w_0$$

- Need to find the values of the **weights** [ $w_0, w_1$ ] ( $= \mathbf{w}$ ) that minimize the sum of squared error over all the training examples:

$$\sum_{j=1}^N (y_j - h_w(x_j))^2 = \sum_{j=1}^N (y_j - (w_1x_j + w_0))^2$$

N : The number of samples  $(x_i, y_i)$  for training



# Univariate Linear Regression

- To minimize the sum  $\sum_{j=1}^N (y_j - (w_1 x_j + w_0))^2$

$$\frac{\partial}{\partial w_0} \sum_{j=1}^N (y_j - (w_1 x_j + w_0))^2 = -2 \sum_{j=1}^N (y_j - (w_1 x_j + w_0)) = 0$$

$$\frac{\partial}{\partial w_1} \sum_{j=1}^N (y_j - (w_1 x_j + w_0))^2 = -2 \sum_{j=1}^N (y_j - (w_1 x_j + w_0)) x_j = 0$$

These equations have a unique, **closed-form solution**:

$$w_0 = (\sum y_j - w_1 (\sum x_j)) / N; \quad w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}$$

Although we could use the Local Search algorithm learned previously, we can calculate the optimal  $w_0, w_1$  using the property that the error is minimized at the point where the derivative equals 0.

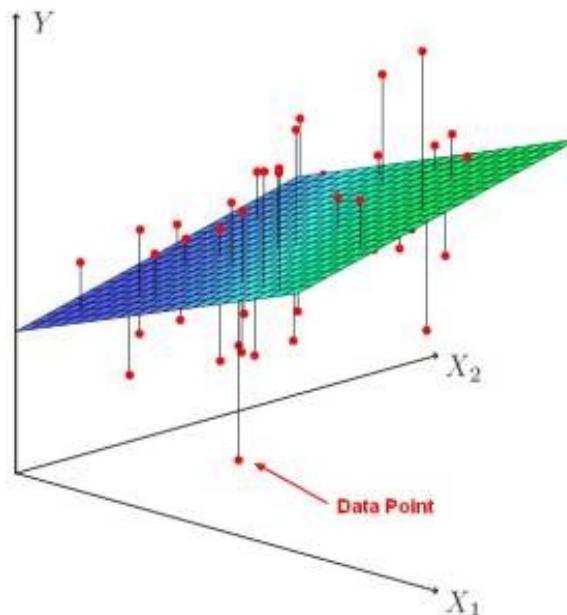
The result of rearranging the second equation above for  $w_1$  and then plugging the resulting  $w_1$  into the equation for  $w_0$  and simplifying.



# MULTIVARIATE LINEAR REGRESSION

# Multivariate Linear Regression

- MLR
  - Until now, we have looked at **Univariate LR** where the independent variable is only **one (a scalar)**.
  - What if there are multiple independent variables?
    - ex: {room: 3 개, Bathroom: 2 개} = 80 만원 ...
  - **We extend Univariate LR to use Multivariate LR! (The operating principle is the same)**



# Multivariate Linear Regression

- Each example  $\mathbf{x}_j$  is an  $d$ -element vector

$$h_{\mathbf{w}}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^T \mathbf{x}_j = \sum_i w_i x_{j,i}$$

where  $x_{j,0} = 1$  is a dummy input attribute (used for the purpose of finding the optimal **intercept** value)

- $h_{\mathbf{w}}(\underline{\mathbf{x}_j})$

$$= w_0 x_{j,0} + w_1 x_{j,1} + w_2 x_{j,2} + \cdots + w_d x_{j,d}$$

$$= \sum_i w_i x_{j,i}$$

$$= \underline{\mathbf{w}}^T \underline{\mathbf{x}_j}$$

$$= [w_0 \quad \dots \quad w_d] \begin{bmatrix} x_{j,0} \\ \vdots \\ x_{j,d} \end{bmatrix}$$

# Multivariate Linear Regression

- The best weight vector  $\mathbf{w}$  that minimizes loss over the examples:
- $\mathbf{w}^* = \arg \min_{\mathbf{w}} \sum_j (y_j - \mathbf{w} \cdot \mathbf{x}_j)^2$
- That is, the goal is to minimize total\_error = minimize  $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$ 
  - Find  $\mathbf{w}^*$  that satisfies  $\mathbf{X}\mathbf{w} - \mathbf{y} = 0$
- The analytical solution is

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

To prepare for the case where the  $\mathbf{X}$  matrix is non-invertible, we calculate  $\mathbf{w}^*$  through the **pseudo-inverse** by multiplying both sides (on the left) by  $\mathbf{X}^T$ , and then multiplying both sides (on the left again) by  $(\mathbf{X}^T \mathbf{X})^{-1}$ .

where  $\mathbf{X}$  is the data matrix of inputs with one  $d$ -dimensional example per row

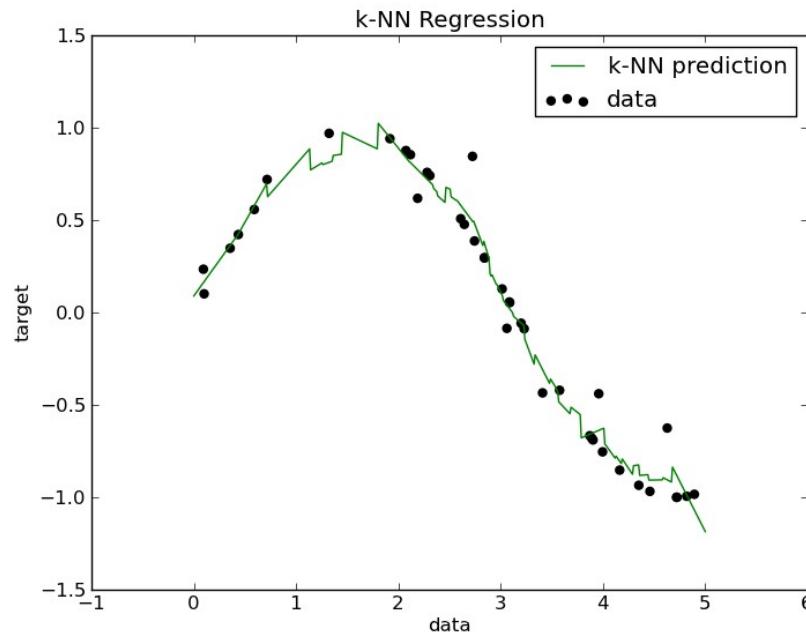
- $j$ -th row contains  $d + 1$  feature values including  $x_{j,0} = 1$
- Note, Univariate LR corresponds to the special case where  $d = 1$ .

# **NEAREST NEIGHBOR MODELS**

# Nearest Neighbor Models (k-NN)

- If the **data is distributed linearly**, it can be predicted using **Linear Regression**.
- However, what if the **data is non-linear**, as shown below?

Let's use k-NN!

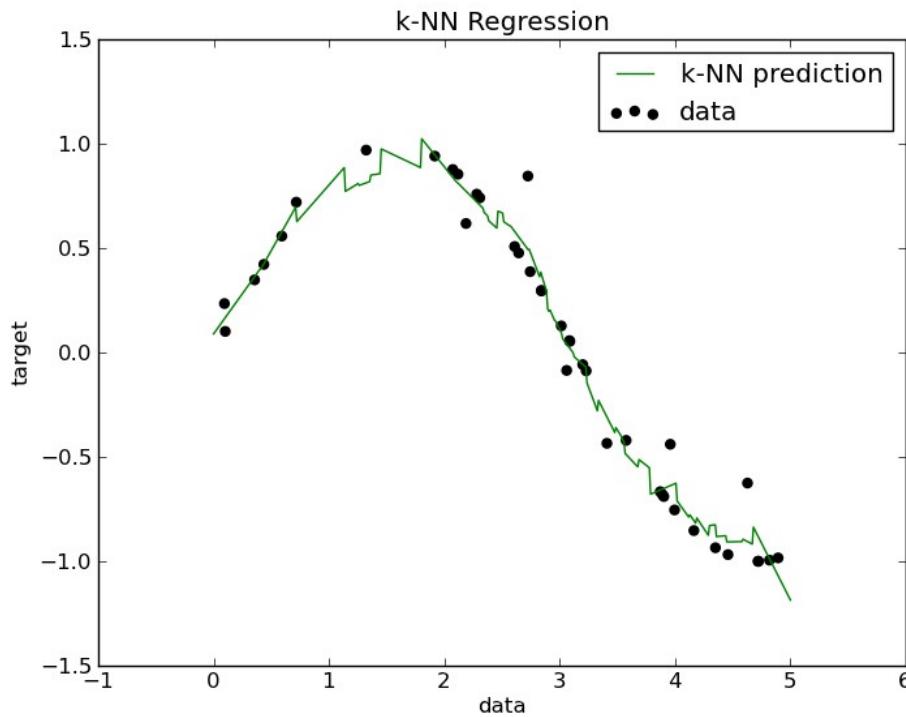


# Nearest Neighbor Models (k-NN)

- NN model
  - **Input:** Training data ( $x_i$ ,  $y_i$ )
  - **Usage:** When unknown data  $x_{new}$  is given, predict what the answer  $y_{new}$  is.
- Method of Operation
  - The NN model does **not** have a separate '**training**' process.
  - When a new input  $x_{new}$  is given, it extracts  $k$  data points from the training data that have  $x$  values **neighboring/adjacent** to  $x_{new}$ , and then calculates  $y_{new}$  by finding the **average** of the  $y_i$  values of those data points.
    - For this, a **method to evaluate** how close/adjacent two  $x_1$  and  $x_2$  are is required.

# Nearest Neighbor Models (k-NN)

- Given a query  $\mathbf{x}_q$  (= Unknown input data), find the  $k$  nearest neighbors  $NN(k, \mathbf{x}_q)$ 
  - $NN(\mathbf{x}_q, k)$ : Used by extracting a set of  $k$  data points that are adjacent/neighboring to  $\mathbf{x}_q$  along the  $x$ -axis.
  - Regression** (Two methods for predicting  $y_{new}$  using  $k$  neighboring values):
    - mean or median of  $NN(k, \mathbf{x}_q)$  or // NN 집합 내의  $y_i$  값의 평균/중간값을 사용하거나
    - solve a linear regression problem on  $NN(k, \mathbf{x}_q)$  // NN 집합 내의 값에 LR 적용

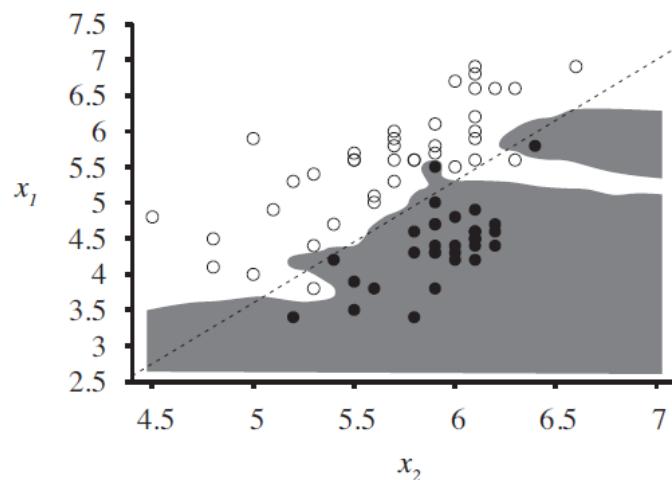


- Green solid line:** The result of applying k-NN to all  $x$  values between [0, 5].
- Black dots:** The  $(x_i, y_i)$  values given beforehand for training.

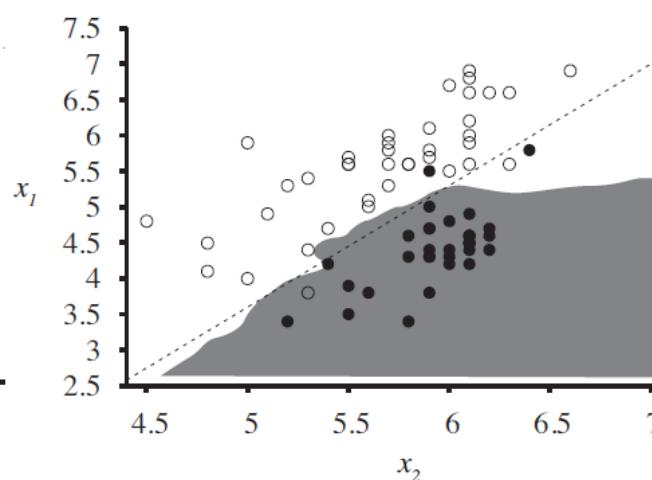
- The k-NN technique does not create a separate model or go through training, but it shows effective performance in predicting values.
- Furthermore, it can handle **non-linear data** more effectively compared to linear regression.

# Nearest Neighbor Models (k-NN)

- Given a query  $\mathbf{x}_q$ , find the  $k$  nearest neighbors  $NN(k, \mathbf{x}_q)$ 
  - Furthermore, the k-NN technique can also be used for **classification** problems.
  - Classification: plurality vote of  $NN(k, \mathbf{x}_q)$  // Majority rule



Overfitting with  $k = 1$



O.K. with  $k = 5$

- In the left graph, which visualizes the (data, answer) pairs, the **data** =  $(\mathbf{x}_1, \mathbf{x}_2)$ , and the **answer** = (white 0 or black 1).
- The goal of a general classification problem is to establish the optimal criteria (e.g., the solid line in the graph) that separates different types of data and to accurately classify unlabeled data.

## Classification using k-NN: Method of Operation

- For all  $(\mathbf{x}_1, \mathbf{x}_2)$ , after selecting the  $k$  closest neighbors, the query is classified as white if the majority of selected data is white and classified as black if the majority is black.
- The problem of **overfitting** occurs when  $k = 1$ , while  $k = 5$  shows considerably smooth classification performance.
- Therefore, determining the value of  $k$  is very important → The optimal  $k$  value can be practically determined by trying various values.

# Nearest Neighbor Models (k-NN)

- Distance Measurement Method
  - The **NN** technique uses 'neighboring or closest data' in its calculation process, and 'neighboring' means a short '**distance**'.
  - Distance can be measured in various ways, and a generalized method called **Minkowski distance** is used to measure distance.
- Distance metric: **Minkowski distance** (also known as  $L^p$  norm)

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = \left( \sum_i |x_{j,i} - x_{q,i}|^p \right)^{1/p}$$

- $p = 2$ : Euclidean distance // The most commonly used measurement standard
- $p = 1$ : Manhattan distance
  - $p = 1$  with Boolean attributes (when each  $x$  value consists only of 0s and 1s): **Hamming distance**

# Nearest Neighbor Models (k-NN)

- Note:
  - Depending on the problem, it may be necessary to **normalize** the data ( $x$ ) values  
**(Normalization:** transforming variables with different ranges so they have similar ranges).
  - Example: Monthly Rent Prediction Problem
    - In a training dataset composed of  $(x, y)$ , where  $y$  is the answer (label)
    - **Data =  $(x_1, x_2)$ ,** where  $x_1$  = house size (square meters), and  $x_2$  = number of bathrooms,
    - If  $x_1$  has values between 0 and 100, and  $x_2$  has values between 0 and 5
    - When calculating the distance between two different data points, the difference in the  $x_1$  values is large, and the difference in the  $x_2$  values is usually small. This means the distance value is primarily determined by  $x_1$ , and  $x_2$  has almost no influence on the prediction.
    - When calculating the distance, it is necessary to **limit the range of each variable's values to a similar level** so that  $x_1$  and  $x_2$  variables contribute equally to the distance calculation.
- **Normalization:**  $x_{j,i} \rightarrow (x_{j,i} - \mu_i)/\sigma_i$ 
  - $\mu_i$ : mean of the values in the  $i$ -th dimension
  - $\sigma$ : standard deviation of the values in the  $i$ -th dimension

# Nearest Neighbor Models (k-NN)

- Various "distance" calculation methods

## Euclidean Distance

This is nothing but the cartesian distance between the two points which are in the plane/hyperplane. [Euclidean distance](#) can also be visualized as the length of the straight line that joins the two points which are into consideration. This metric helps us calculate the net displacement done between the two states of an object.

$$\text{distance}(x, X_i) = \sqrt{\sum_{j=1}^d (x_j - X_{i,j})^2}$$

## Manhattan Distance

[Manhattan Distance](#) metric is generally used when we are interested in the total distance traveled by the object instead of the displacement. This metric is calculated by summing the absolute difference between the coordinates of the points in n-dimensions.

$$d(x, y) = \sum_{i=1}^n |x_i - y_i|$$

## Minkowski Distance

We can say that the Euclidean, as well as the Manhattan distance, are special cases of the [Minkowski distance](#).

$$d(x, y) = \left( \sum_{i=1}^n (x_i - y_i)^p \right)^{\frac{1}{p}}$$

From the formula above we can say that when  $p = 2$  then it is the same as the formula for the Euclidean distance and when  $p = 1$  then we obtain the formula for the Manhattan distance.

# Nearest Neighbor Models (k-NN)

- Example of KNN code for classifying the input value ( $p$ ) into Group 0 or Group 1

```
distance=[]
for group in points:
    for feature in points[group]:
        #calculate the euclidean distance of p from training points
        euclidean_distance = math.sqrt((feature[0]-p[0])**2 +(feature[1]-p[1])**2)
        # Add a tuple of form (distance,group) in the distance list
        distance.append((euclidean_distance,group))

# sort the distance list in ascending order
# and select first k distances
distance = sorted(distance)[:k]
freq1 = 0 #frequency of group 0
freq2 = 0 #frequency og group 1

for d in distance:
    if d[1] == 0:
        freq1 += 1
    elif d[1] == 1:
        freq2 += 1

return 0 if freq1>freq2 else 1
```

Calculates the Euclidean distance between the input value ( $p$ ) and all points in the dataset.

Extracts  $k$  data points that have the minimum distance.

Counts the number of data points included in Group 0 and the number of data points included in Group 1 among the  $k$  data points.

# End