Search Algorithms: Object-Oriented Implementation (Part D)

Contents

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

Summary of Implementation So Far

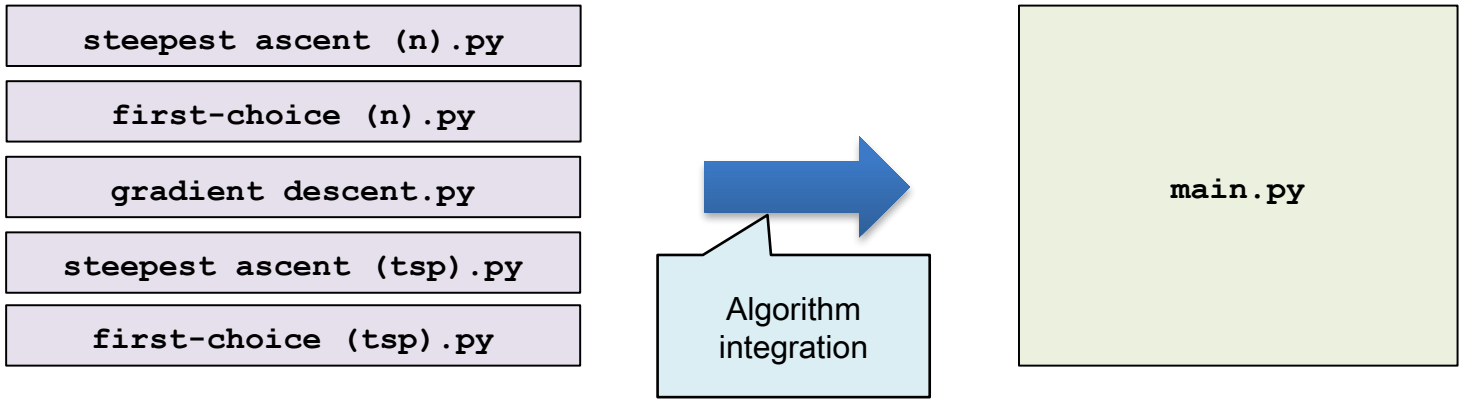
- Search Algorithm (C) / Search Tool v2 Implementation Status
  - problem.py : Definition of the class for *variables/functions related to the problem's solution*.
    - `Problem` Class definition : Includes content common regardless of the problem type (e.g., saving the solution, saving `numEval`, etc.)
    - 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

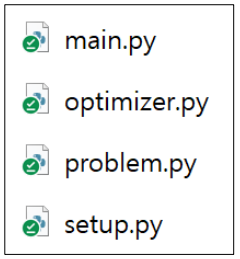
Code Improvement Idea

- 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!



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.

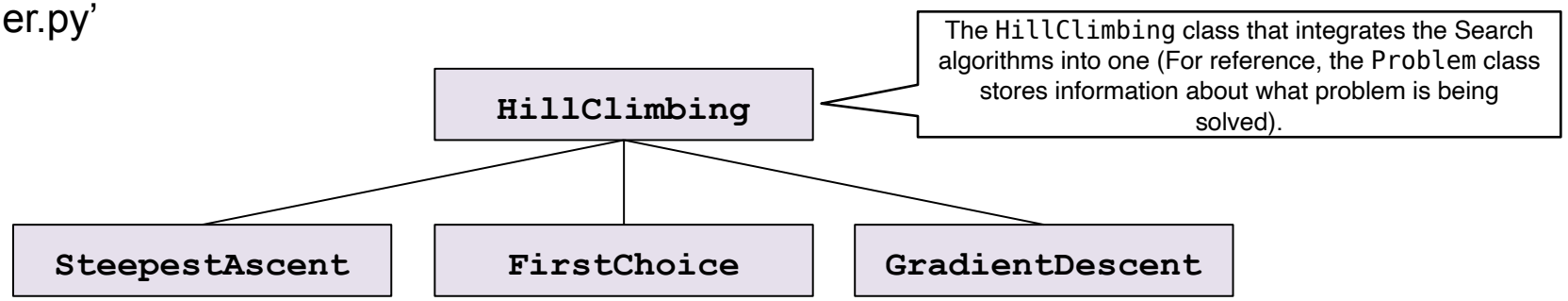
  - `problem.py` : Defines and differentiates classes based on the **nature of the problem.**
    - Definition of the `Problem` class (inheriting from `Setup`).
    - Definition of the `Numeric` class (inheriting from `Problem`).
    - Definition of the `TSP` class (inheriting from `Problem`).

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

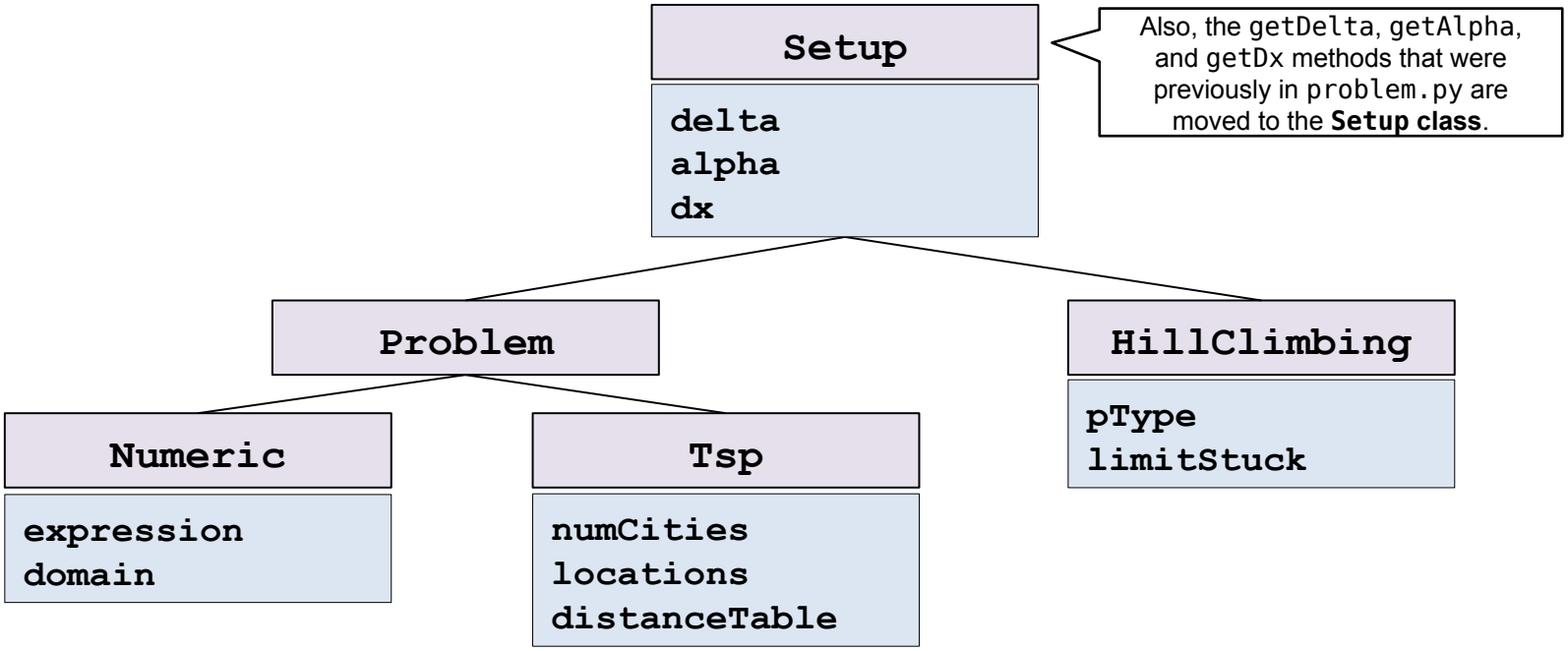
- What if we make **`alpha`** and **`dx`** variables of **`GradientDescent`**, and **`delta`** a variable of **`HillClimbing`**?
  - Not a good idea because they are already variables of the subclass **`Numeric`**

- 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

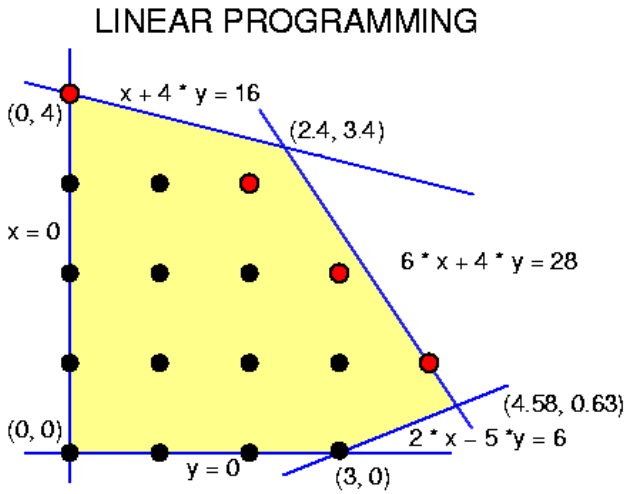
- **`2. selectProblem()`**:
  - Asks the user to choose the type of problem to be solved
  - Creates a **`Problem`** object **`p`** of the right type
  - Sets the variables of the corresponding subclasses of **`Problem`**
  - Returns **`p`** and **`pType`** (an integer indicating the problem type)

- **`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.



- Continuous ...em
  - Every ...n 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

http://www.cse.chalmers.se/edu/year/2011/course/TDA206_Discrete_Optimization/

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 execture 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`**