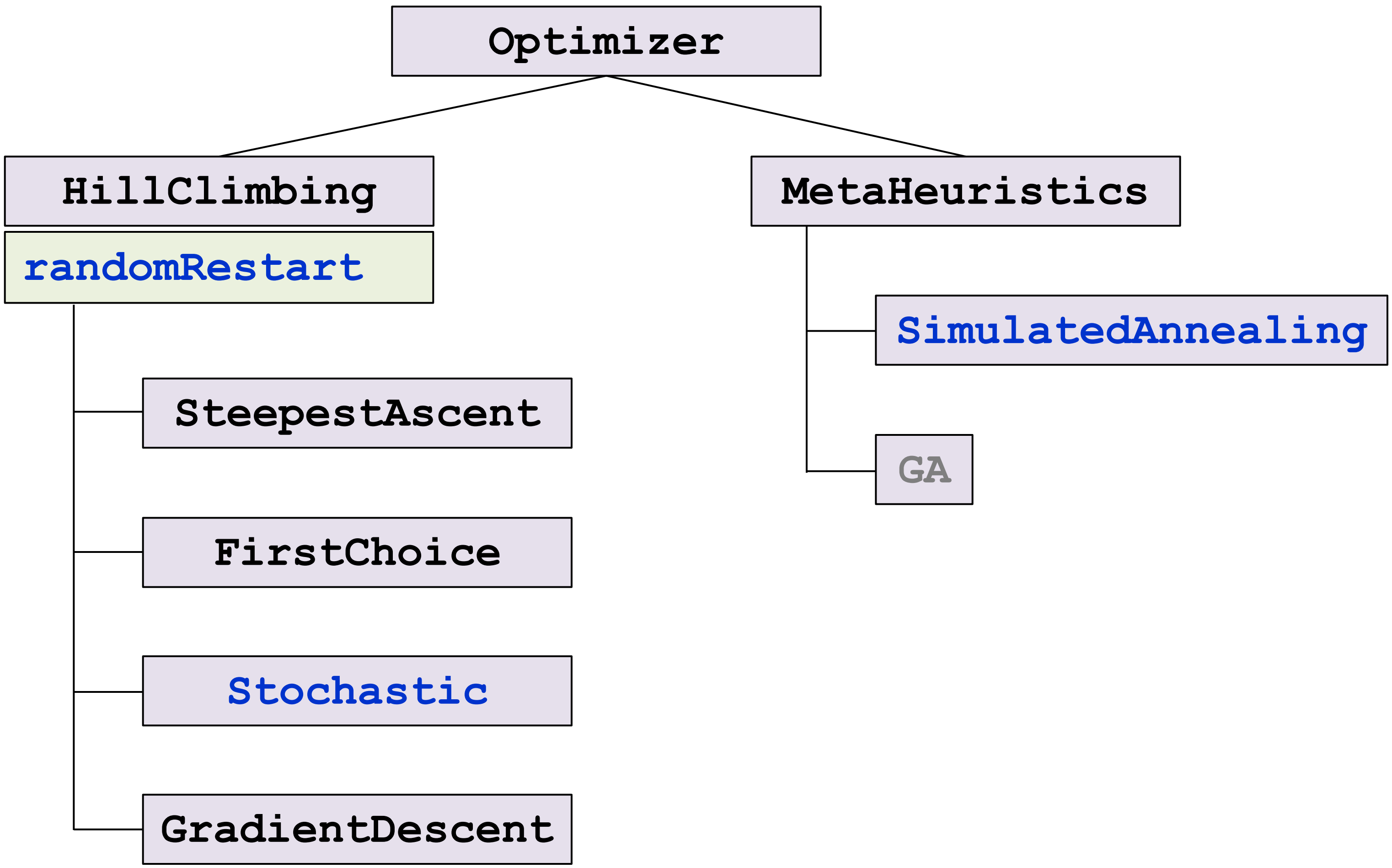
Search Algorithms: Object-Oriented Implementation (Part E)

Contents

Adding More Algorithms and Classes

- We add three more search algorithms: stochastic hill climbing, random-restart, and simulated annealing
  - This requires a more general class hierarchy to cover various algorithms
  - We define a new class **`Optimizer`** and let it have two subclasses **`HillClimbing`** and **`MetaHeuristics`**
  - **`HillClimbing`** has four child classes, each for a specific hill climber
    - The random-restart algorithm becomes a method of **`HillClimbing`**, which can be inherited to all the child hill climbers
  - **`MetaHeuristics`** has two child classes, **`SimulatedAnnealing`** and **`GA`** (to be implemented later)

Adding More Algorithms and Classes



Adding More Algorithms and Classes

- Notice that **`randomRestart`** is a wrapper around any hill-climbing algorithm, which runs the algorithm for a given number of times
  - Each hill climber can have not only a **`run`** method for its own algorithm but also a **`randomRestart`** method as a wrapper that keeps calling the run method
  - Instead of having the same **`randomRestart`** method in duplicate in all the individual hill-climber subclasses, we make it a method of the parent class **`HillClimbing`** so that it can be inherited to all the subclasses

Adding More Algorithms and Classes

- When running any of the hill-climbing algorithms
  - An object **`alg`** is created for one of the four hill-climber subclasses chosen by the user (e.g., **`alg = steepestAscent()`**)
  - Then, the **`randomRestart`** method is called from that object (**`alg.randomRestart`**)
    - See the **`createOptimizer`** and **`conductExperiment`** function of the main program
  - Notice that **`randomRestart`** keeps calling the **`self.run`** method that is the algorithm of the chosen hill-climber

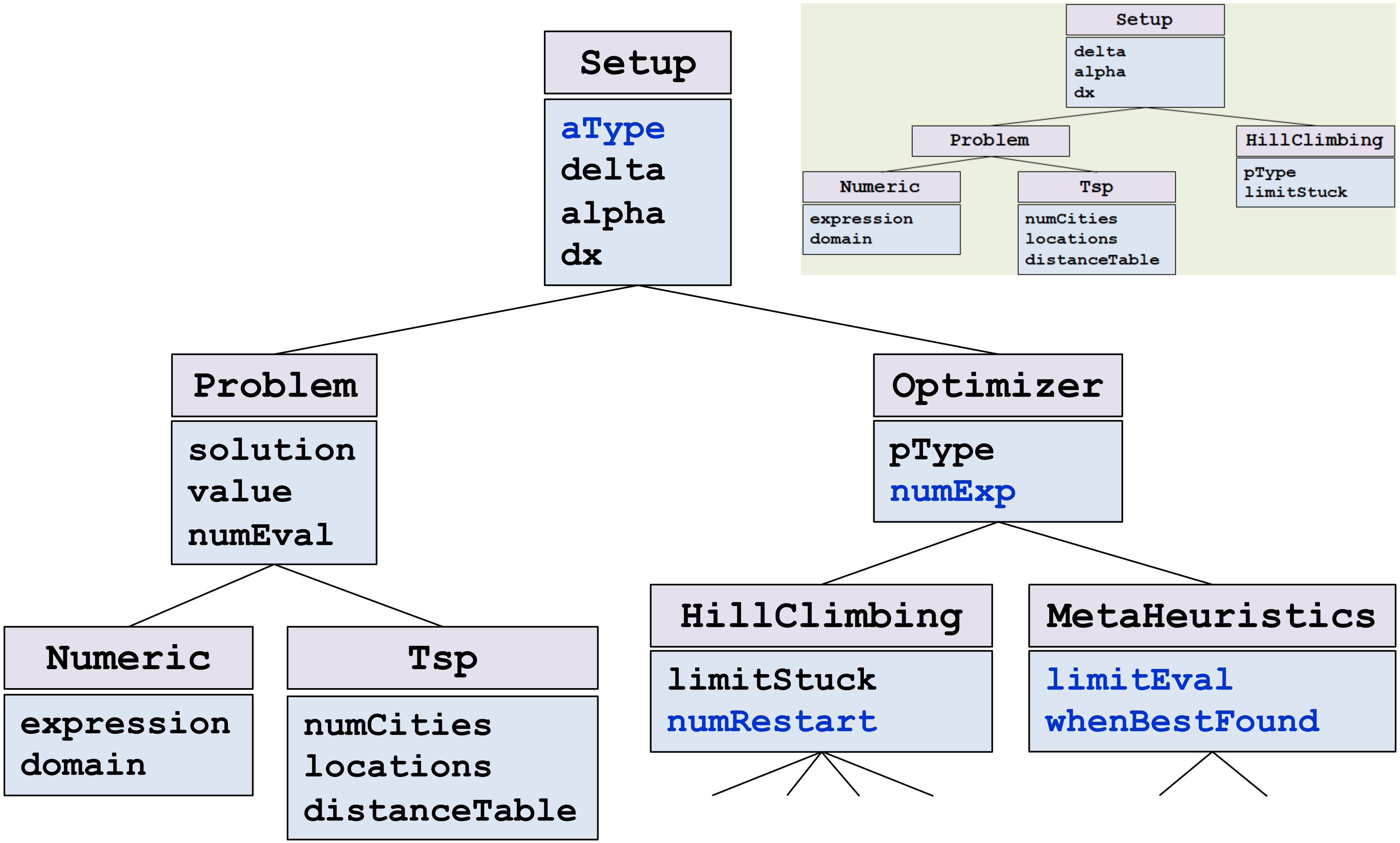
- In the case of metaheuristic algorithms, simply the class object of a chosen algorithm is created and run

Adding More Algorithms and Classes

- Due to the variety of algorithms and related parameters, the user interface is revised to get setup information from a file

- There are four variables newly added to **`Optimizer`** class hierarchy
  - **`numExp`**: total number of experiments to be conducted (to **`Optimizer`**)
  - **`limitEval`**: total number of evaluations until termination for metaheuristic algorithms (to **`MetaHeuristics`**)
  - **`whenBestFound`**: the number of iterations taken until the best solution has first been found (to **`MetaHeuristics`**)
  - **`numRestart`**: number of random restarts for the random-restart algorithm (to **`HillClimbing`**)

Adding More Algorithms and Classes



Adding More Algorithms and Classes

- **`aType`** is newly added to **`Setup`**
  - It is referenced by the revised **`report`** method of **`Problem`**
  - It is also referenced by **`displaySetting`** of **`HillClimbing`**

- Because of the new **`optimizer`** class hierarchy, **`pType`** has moved up from **`HillClimbing`** to **`Optimizer`** because it is referenced by the revised **`displaySetting`** method of **`Optimizer`**
  - A variable **`v`** is better placed at the lowest possible class c such that **`v`** is used in that class or in its subclasses

Adding More Algorithms and Classes

- Notice that an instance of an object in a class hierarchy is created for a leaf class most of the times
  - Things in the higher classes are all inherited unless otherwise specified
  - To inherit methods of the same names, it should be stated explicitly (e.g., **`Optimizer.setVariables(self, parameters)`**)
    - Otherwise, the methods in the upper classes will simply be overridden
  - Whenever a new instance is created, all the variables up to the base class must be set to appropriate values by calling the relevant methods such as **`setVariables`**

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
  - Experimental settings and other information should be read from a file provided by the user

The New Main Program

- **`main()`**:
  - Reads information from a file and creates a problem **`p`** (**`Problem`** object) to be solved and an optimizer **`alg`** (**`Optimizer`** object) to be used (**`readPlanAndCreate`**)
  - 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`**)

The New Main Program

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

- **`readValidPlan()`**:
  - Reads setup information from a file and stores them in **`parameters`** (**`readPlan`**)
  - Keeps querying the user if gradient descent is chosen for TSP
  - Returns **`parameters`**

The New Main Program

- **`readPlan()`**:
  - Obtains a file name from the user
  - Prepares a dictionary variable **`parameters`** to store the information
  - Fills out **`parameters`** dictionary by reading the given file
    - All the information are numeric values except one file name (the file containing specifics of the target problem)
    - When reading the file, lines beginning with '#' are all skipped (**`lineAfterComments`**)
  - Returns **`parameters`**

The New Main Program

- **`lineAfterComments()`**:
  - Skips the lines beginning with the symbol ‘#’ and returns the first line beginning with no ‘#’

- **`createProblem(parameters)`**:
  - Creates a **`Numeric`** or **`Tsp`** object **`p`** depending on the type of problem chosen
  - Sets some relevant variables of **`p`** in the class hierarchy with 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`**
  - 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`**)
  - Repeats experiment multiple times (**`numExp`**) if requested and collects the results in a few local variables
  - Stores the final summary 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
  - **`bestSolution`**: best solution found in n different experiments
  - **`bestMinimum`**: objective value of **`bestSolution`**
  - **`avgMinimum`**: average objective value of the best solutions obtained from n experiments
  - **`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

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

- 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

Changes to 'Problem' Class

  - Those in the subclasses print messages specific to the type of problem just solved

- The **`reportNumEvals`** method prints out the total number of evaluations regardless of the problem type
  - However, it does nothing when the algorithm used is simulated annealing or GA because the number of evaluations for them is predetermined
  - It is separated from **`report`** because we want the result messages printed out in some appropriate order when mixed together with the messages generated from the subclasses
  - Call to **`reportNumEvals`** is made within **`report`** of the subclasses at its last line

‘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 of the main program

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

'Optimizer' Class

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

```
HillClimbing
-----------------
limitStuck
numRestart
-----------------
__init__
setVariables
displaySetting
randomRestart
```

```
MetaHeuristics
-----------------
limitEval
whenBestFound
-----------------
__init__
setVariables
getWhenBestFound
displaySetting
```

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 part of printing the mutation step size is moved up to **`Optimizer`**
  - Now it prints out setting information related only to the variables of its own class **`HillClimbing`**

Changes to 'HillClimbing' Class

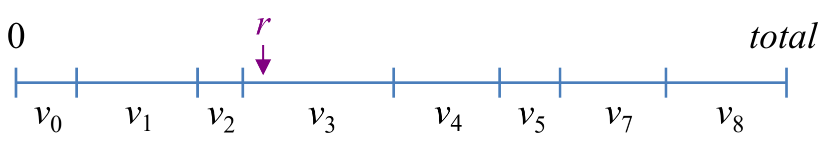
- A new method **`randomRestart`** is added as described before
  - It keeps calling **`self.run`** for a given number of times (**`self._numRestart`**) and stores the best solution found (**`p.storeResult`**)

Stochastic Hill Climbing

- 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 quality
  - The algorithm is implemented as the **`run`** method as before

Stochastic Hill Climbing

- **`stochasticBest(self, neighbors, p)`**:
  - Obtains a list of evaluation values of **`neighbors`** (**`valuesForMin`**)
    - Here, smaller values are better
  - Converts the list to the one in which larger values are better (**`valuesForMax`**)
    - Each original value is subtracted from a large enough value (the maximum plus one to avoid zero)
  - Chooses at random from **`valuesForMax`** with probability proportional to its value



  - Returns the chosen neighbor together with its original evaluation value

Simulated Annealing

- There is one variable **`numSample`** storing the number of samples used to heuristically determine an initial temperature
  - It is currently preset to 100
- **`run()`**:
  - Starts from a random initial point
  - While the algorithm runs for **`limitEval`** iterations or stops when the temperature is zero, it uses another variable **`whenBestFound`** to record when the best-so-far solution has first been found
  - The temperature decreases every iteration according to an annealing schedule (**`self.tSchedule(t)`**)
  - The initial temperature is heuristically determined so that the probability of accepting a worse neighbor becomes 0.5 initially (**`self.initTemp(p)`**)

Simulated Annealing

- The probability of accepting a worse neighbor is `exp(-dE/t)`, where `dE` is the difference of the evaluation values and `t` is the temperature

- `initTemp(self, p)`:
  - Takes k (= `self._numSample`) random samples and their neighbors from the domain of problem `p`
  - Calculates the average `dE` of their differences
  - Calculates the temperature `t` such that `exp(-dE/t)` $= 0.5$, and returns `t`
- `tSchedule(self, t)`:
  - Calculates the next temperature using a simple formula, and returns it