Search Algorithms: Object-Oriented Implementation (Part C)

Contents

Part C-1

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.

variables

functions

Defining Classes

problem.py source code define three classes

Problem

solution
value
numEval

__init__
storeResult
report

Numeric

expression
domain
delta = 0.01

__init__
setVariables
getDelta (new)
randomInit
evaluate
mutants
mutate
randomMutant
describe
report
coordinate

Tsp

numCities
locations
distanceTable

__init__
setVariables
calcDistanceTable
randomInit
evaluate
mutants
inversion
randomMutant
describe
report
tenPerRow

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**

```python
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
```

```python
    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: {0:,}".format(
```

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 <u>three variables</u> 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

```python
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 <u>three variables</u> 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 **`dis-playSetting`** 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 100×100 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 <u>storing the result of search</u>:
  - **`solution`**: final solution found by the search algorithm, x*
  - **`value`**: its objective value, f(x*)
  - **`numEval`**: total number of evaluations taken for the search

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

  - 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 solusion report of `Problem.report` are shared between `Numeric.report` and `Tsp.report` with slight changes for respective problem.

```
def report(self): # Numeric.report
    print()
    print("Solution found:")
    print(self.coordinate())  # Convert list to tuple
    print("Minimum value: {0:,.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: {0:,}".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: {0:,}".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 regard-less 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 va
    # 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()
```

```
def main():
    # Create a Problme 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()
```

first-choice(n).py

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

main()
```

```
def main():
    # Create a Problme 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 Problme 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)

ex) Steepest Ascent for Numeric Optimization

```
def main():
    # Create a Problme 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

ex) Steepest Ascent for Numeric Optimization

```
def main():
    # Create a Problme 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`**)

ex) Steepest Ascent for Numeric Optimization

```
def main():
    # Create a Problme 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

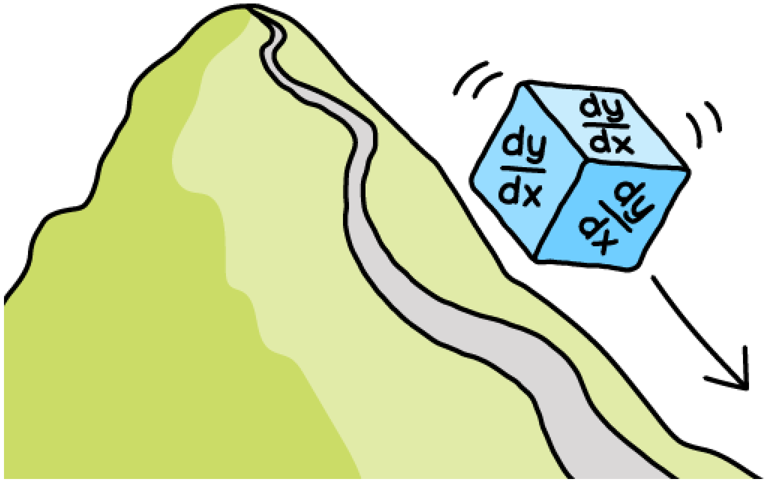
- There were two versions of **`mutants`** for the steepest-ascent hill climbing: one for numeric optimization and the other for TSPs
  - They are now migrated to the **`Numeric`** and **`Tsp`** classes

- 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

Part C-2

Next



Solution for solving continuous variable problems

GRADIENT DESCENT ALGO.