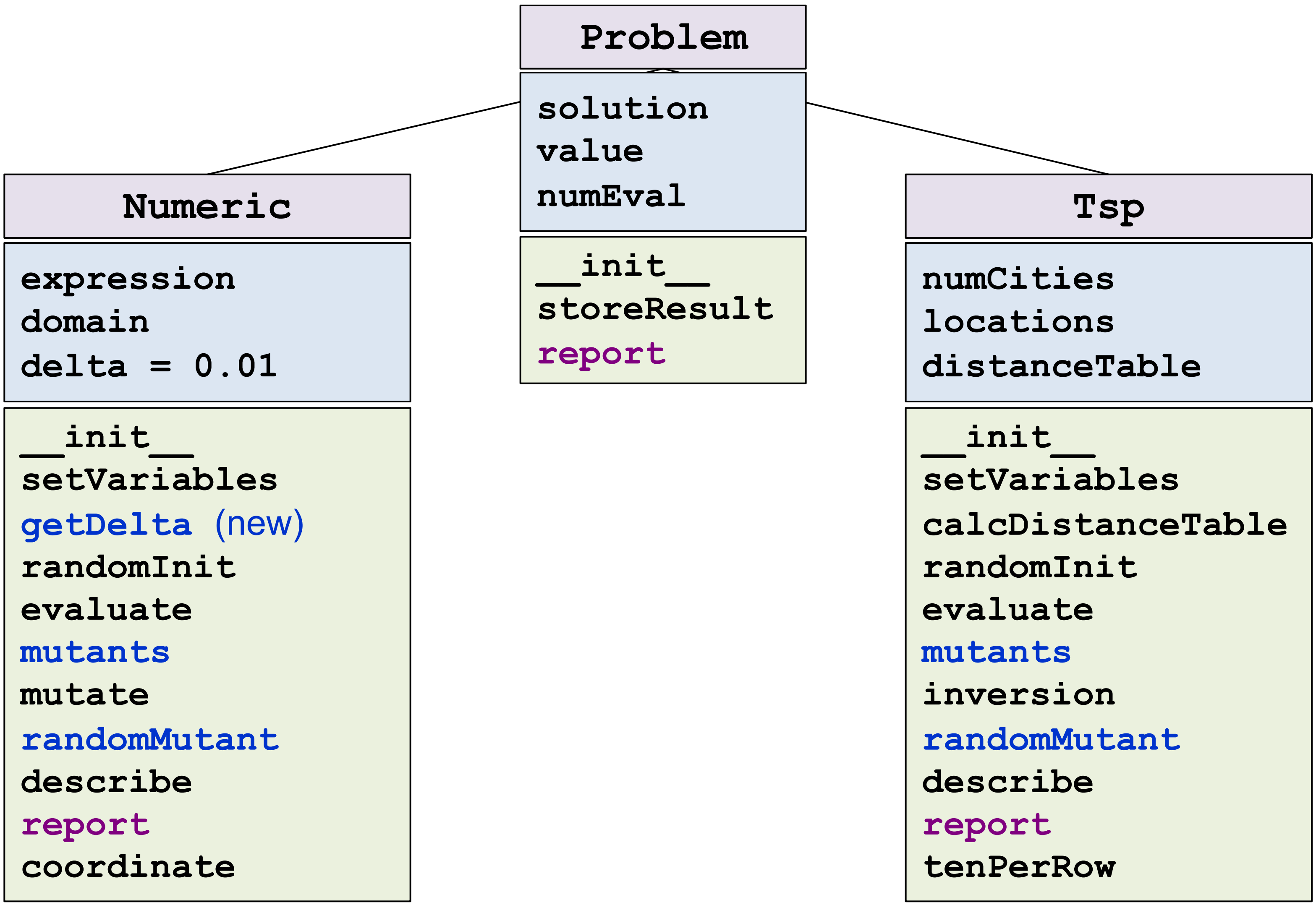
Search Algorithms: Object-Oriented Implementation (Part C)

preliminary version

Contents

Defining ‘Problem’ Class



Defining ‘Problem’ Class

- The base class `Problem` has three variables for storing the result of search:
  - `solution`: final solution found by the search algorithm
  - `value`: its objective value
  - `numEval`: total number of evaluations taken for the search

  However, the way to report the result is different depending on the type of problem solved

- The values of the two variables `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

Defining 'Problem' Class

- `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 inherit 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 ‘Problem’ Class

- We store **`Problem`** 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

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 corresponding values in the relevant class variables
  - Previously this was done by the function `createProblem`

- 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

Adding Gradient Descent

- Gradient descent is the same as the steepest-ascent except the way a next point is created from the current point
  - 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 move to a next point
    - n evaluations are needed to calculate partial derivatives in all the dimensions, where n is the dimension of the objective function
    - One additional 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

- Two variables are newly added to the **`Numeric`** subclass:
  - **`alpha`**: update rate for gradient descent
    - 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`**

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$$

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

Adding Gradient Descent

- Also, five 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`**

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \cdots, \frac{\partial f(\boldsymbol{x})}{\partial x_d} \right)^T$$

    - 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`**)

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

Adding Gradient Descent

- `gradient(self, x, v)`
  - Calculates partial derivatives at `x`

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = \frac{f(\boldsymbol{x}') - f(\boldsymbol{x})}{\delta}$$

$$\boldsymbol{x}' = (x_1, \ldots, x_{i-1}, x_i + \delta, x_{i+1}, \ldots, x_d)^T$$

  - Returns the gradient $\nabla f(\boldsymbol{x})$
- `isLegal(self, x)`
  - Checks if `x` is within the domain
- `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