Search Algorithms: Object-Oriented Implementation (Part B)

Contents

Implementing Hill-Climbing Algorithms

- Our eventual goal is to implement an optimization tool consisting of various search algorithms

- We begin to implement two hill climbing algorithms each for two different types of problems:
  - 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)

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`**)
  - 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

- **`createProblem()`**:
  - Gets a file name from the user and reads information from the file
    - Function expression – saved as a string
    - 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
    - Variable names
    - Lower bounds
    - Upper bounds

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 *n* (= # of variables) 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
- **`randomInit(p)`**:
  - Given a problem **`p`**, returns a point randomly chosen within **`p`**'s domain

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
    - A global variable **`NumEval`** is employed as a counter to record the total number of evaluations
- **`mutants(current, p)`**:
  - Returns $2n$ neighbors of **`current`**
  - Each neighbor is made by copying **`current`**, randomly choosing a variable of **`p`**, and then altering its value (**`mutate`**) by both adding and subtracting **`DELTA`**
    - **`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
- **`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`**
- **`desplaySetting()`**:
  - Shows that the steepest-ascent hill climbing has been used as the search algorithm
  - Displays the step size of the axis-parallel mutation

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

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

First-Choice Hill Climbing for Numerical Optimization

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

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`**

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

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

Introducing 'numeric' Module

- 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
- Only a few functions remain in the main programs of SAHC-N and FCHC-N after the code migration

SAHC-N

```
main
steepestAscent
mutants
bestOf
displaySetting
```

FCHC-N

```
main
firstChoice
randomMutant
displaySetting
```

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

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 without much change
  - We see that the functions **`main`**, **`steepestAscent`**, and **`bestOf`** can be reused without any change at all
  - We also see that **`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 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

- **`calcDistanceTable(numCities, locations)`**:
  - Calculates an $n \times n$ matrix of pairwise distances based on **`locations`** (n = **`numCities`**)
- **`randomInit(p)`**:
  - Returns a randomly shuffled list of ids of the cities in **`p`**
- **`Evaluate(current, p)`**:
  - Calculates the tour cost of **`current`** by looking at the distance matrix given in **`p`**
- **`inversion(current, i, j)`**:
  - Makes a copy of **`current`**, inverts its subsection from **`i`** to **`j`**, and returns the mutant
  - This function takes the role of **`mutate`** for numerical optimization

Steepest-Ascent Hill Climbing for TSP

- **`describeProblem(p)`**:
  - Prints the number of cities in **`p`**, followed by the city locations, five locations per line
- **`displayResult(solution, minimum)`**:
  - Displays **`solution`** (the best tour found) (**`tenPerRow`**), **`minimum`** (its evaluation value), and the total number of evaluations
- **`tenPerRow(solution)`**:
  - Prints city ids, ten ids per row

Steepest-Ascent Hill Climbing for TSP

- Below, we describe how mutants in the main program is implemented differently for TSPs than numerical optimization

- **`mutants(current, p)`**:
  - Returns n neighbors of **`current`** (n = 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

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 subsection beginning from **`i`** and ending at **`j`** (**`inversion`**), where **`i`** and **`j`** are chosen randomly