Search Algorithms: Object-Oriented Implementation (Part F)

Contents

Adding Genetic Algorithm

- We add a subclass **`GA`** under **`MetaHeuristics`** in the **`Optimizer`** class hierarchy
  - Accordingly, many problem-dependent methods for **`GA`** are added to the **`Problem`** class
- **`GA`** has seven variables:
  - **`popSize`**: population size
  - **`uXp`**: swap probability for uniform crossover used in numerical optimization
  - **`mrF`**: multiplication factor to $(1/l)$ for bit-flip mutation used in numerical optimization
  - **`XR`**: crossover rate for the permutation code for solving TSP

GA
popSize uXp mrF XR mR pC pM
__init__ setVariables displaySettings run evalAndFindBest selectParents selectTwo binaryTournament

Adding Genetic Algorithm

- `mR`: mutation rate for the permutation code for solving TSP
- `pC`: crossover probability (`uXp` or `XR`)
- `pM`: mutation probability (`mrF` or `mR`)

For binary code   For permutation code

- The `setVariables` method, after setting `popSize`, sets the variables (`pC`, `pM`) to either the values of (`uXp`, `mrF`) or those of (`XR`, `mR`) depending on whether the problem at hand is numerical optimization or TSP, respectively
  - While (`pC`, `pM`) are the parameters of the search algorithm, the user only gives the values of (`uXp`, `mrF`) or (`XR`, `mR`)

Adding Genetic Algorithm

- **`displaySetting`** shows the population size and the parameter values used by the genetic operators
  - **`resolution`**, **`uXp`**, and **`mrF`** for a numerical optimization problem
  - **`XR`** and **`mR`** for a TSP

- The genetic algorithm itself is implemented as the **`run`** method of the class
  - Among the many different genetic algorithms, our implementation is just one of them
  - The population is a list of individuals
  - An individual is a list of a pair: [*fitness*, *chromosome*]
  - The **`run`** method makes calls to some methods of **`Problem`** class and other methods within the **`GA`** class

Adding Genetic Algorithm

- `run(self, p)`:
  - Generates an initial population randomly (`p.initializePop`)
  - Evaluates individuals in the initial population and identifies the best one (`self.evalAndFindBest`)
  - Until termination, keeps applying the genetic operators (`self.selectParents`, `p.crossover`, `p.mutation`), updating the population, and updating the best-so-far individual (`self.evalAndFindBest`)
  - Converts the best individual found to a form containing only the solution part (`p.indToSol`)
  - Stores the best solution (`p.storeResult`)

Adding Genetic Algorithm

- **`evalAndFindBest(self, pop, p)`**:
  - Evaluates each individual in the population **`pop`** (**`p.evalInd`**), identifies the best one, and returns it
- **`selectParents(self, pop)`**:
  - Performs binary tournament selection twice (**`self.selectTwo`**, **`self.binaryTournament`**) and returns the selected parents
- **`selectTwo(self, pop)`**:
  - Selects two random individuals in **`pop`** and returns them
- **`binaryTournament(self, ind1, ind2)`**:
  - Returns the winner between **`ind1`** and **`ind2`**

Changes to 'Setup' Class

- A variable named **`resolution`** is newly added
  - It represents the length of the binary string for each variable for a numeric optimization problem
  - It is referenced by the methods in both the classes **`Optimizer`** and **`Problem`**
- The **`setVariables`** method is accordingly changed

- Changes to the main program is minimal
  - **`readPlan`** is revised to read in more parameter values such as the population size, crossover rate, mutation rate, etc.
  - **`createOptimizer`** is revised to include GA as its 6th optimizer

Changes to ‘Problem’ Class

- Many methods are added to the classes **`Numeric`** and **`Tsp`** to support operations needed to conduct the search by genetic algorithm
- We first describe the five methods that are commonly added to both **`Numeric`** and **`Tsp`**
  - **`initializePop(self, size)`**:
    - Makes a population of given **`size`** with randomly generated individuals, and returns it
    - In **`Numeric`**, the individual chromosome for GA is represented by a binary string that is different from that used by other algorithms (**`self.randBinStr`**)
    - In **`Tsp`**, the individual chromosome for GA is represented by permutation code that is also used by other search algorithms (**`self.randomInit`**)

Changes to ‘Problem’ Class

- **`evalInd(self, ind)`**:
  - Evaluates the chromosome of **`ind`** and records the fitness value (**`self.evaluate`**)
  - In **`Numeric`**, however, the binary string must be decoded before it can be evaluated (**`self.decode`**)
- **`crossover(self, ind1, ind2, pC)`**:
  - Performs crossover to parents and returns the resulting children
  - In **`Numeric`**, a uniform crossover is performed interpreting **`pC`** as the swap probatility **`uXp`** (**`self.uXover`**)
  - In **`Tsp`**, an ordered crossover is performed interpreting **`pC`** as the crossover rate **`XR`** (**`self.oXover`**)

Changes to ‘Problem’ Class

- **`mutation(self, ind, pM)`**:
  - Performs mutation to **`ind`** and returns it
  - In **`Numeric`**, a bit-flip mutation is performed interpreting **`pM`** as the factor **`mrF`** to adjust the mutation rate
  - In **`Tsp`**, an inversion operation is performed interpreting **`pM`** as the mutation rate **`mR`** (**`self.inversion`**)
- **`indToSol(self, ind)`**:
  - Converts an individual to a form containing only the solution part
  - In **`Numeric`**, the chromosome is decoded and then returned (**`self.decode`**)
  - In **`Tsp`**, just the chromosome part of **`ind`** is returned

Changes to 'Problem' Class

- We now describe the methods that are added only to **`Numeric`**
  - **`randBinStr(self)`**:
    - Generates a random binary string of a predetermined length (**`self._resolution`**), and returns it
  - **`decode(self, chromosome)`**:
    - Decodes each variable in **`chromosome`** to its decimal value (**`self.binaryToDecimal`**), concatenates them to a solution form, and returns it
  - **`binaryToDecimal(self, binCode, l, u)`**:
    - Decodes **`binCode`** to a decimal value taking the domain and resolution into account, and returns it

Changes to 'Problem' Class

- **`uXover(self, chrInd1, chrInd2, uXp)`**:
  - Performs uniform crossover to two chromosomes **`chrInd1`** and **`chrInd2`**, and returns the resulting chromosomes

- We now describe one method that is added only to **`Tsp`**
  - **`oXover(self, chrInd1, chrInd2)`**:
    - Performs ordered crossover to two chromosomes **`chrInd1`** and **`chrInd2`**, and returns the resulting chromosomes

Experiments

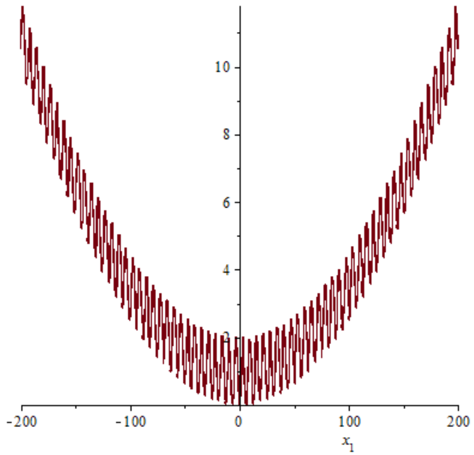
- We solve a few numerical optimization problems and TSPs using various search algorithms made available in our optimization tool and compare their performances
- We try three numerical optimization problems all of which are five dimensional with the search space of $-30 \leq x_i \leq 30$ for $1 \leq i \leq 5$
  - Convex function:

$$(x_1 - 2)^2 + 5(x_2 - 5)^2 + 8(x_3 + 8)^2 + 3(x_4 + 1)^2 + 6(x_5 - 7)^2$$

  - Griewank function:

$$1 + \frac{1}{4000}\sum_{i=1}^{5} x_i^2 - \prod_{i=1}^{5} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

with a global minimum of 0 at all zeros



One-dimensional Griewank function

Experiments

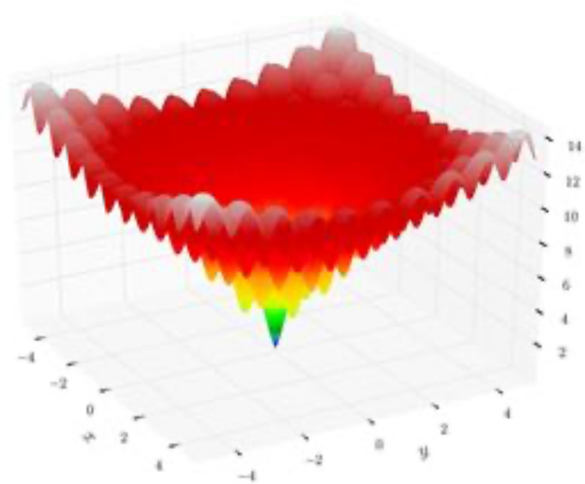
– Ackley function:

$$20+e-20\exp\left(-\frac{1}{5}\sqrt{\frac{1}{5}\sum_{i=1}^{5}x_i^2}\right)-\exp\left(\frac{1}{5}\sum_{i=1}^{5}\cos(2\pi x_i)\right)$$

with a global minimum of 0 at all zeros

- We also try three versions of TSPs with different numbers of cities located in a 100×100 square
  - We use TSP-N as the name of a TSP with N cities, where N is 30, 50, and 100

Two-dimensional Ackley function



Experimental Setting

- Step size of mutation for the three hill climbers steepest-ascent, first-choice, and stochastic: 0.01
- For the gradient descent algorithm:
  - Update rate: 0.01
  - Size of dx for calculating gradient: 10^{-4}

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

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

- For the two hill climbers first-choice and stochastic:
  - Consecutive iterations allowed for no improvement: $1{,}000$
- For the two metaheuristic algorithms:
  - The total number of evaluations until termination: $500{,}000$

Experimental Setting

- Population size of GA: 100
- GA for numerical optimization:
  - The length of binary chromosome per variable: 10
  - Swap probability for uniform crossover: 0.2
  - Multiplication factor to $1/l$ for mutation (l: length of chromosome): 1
- GA for TSP:
  - Crossover (ordered crossover) rate: 0.5
  - Mutation (inversion) rate: 0.2

Experimental Results

- The numbers shown in the table are the averages of 10 experiments
  - All the hill climbers are randomly restarted by 10 times in each experiment
- The four numbers in each cell represent the following:
  - Average objective value (left top)
  - Best objective value found (left bottom)
  - Average number of evaluations (right top)
  - Average iteration of finding the best solution (right bottom) (only for GA and Simulated Annealing)
- Bold-faced letters indicate the best result among different optimizers
- For TSPs, the results are compared with those obtained by the nearest-neighbor algorithm, which starts from a random city and keeps visiting the one that is the closest

Experimental Results

- Convex function:
  - All the algorithms except GA found the optimal solution
  - Gradient descent reaches the optimum the fastest (SA is faster but it does not terminate automatically)
  - First-choice is faster than steepest-ascent
- Griewank function:
  - GA performs much better than the others
  - SA performs worse than the hill climbers
- Ackley function:
  - GA is much better than the others
  - SA performs worse than the hill climbers

Experimental Results

- TSPs:
  - Steepest-ascent is worse than nearest-neighbor
  - Stochastic shows the best average performance with TSP-50
    - But it took too many iterations to solve TSP-100
  - SA shows the overall best performance
  - GA does not show any competitive performance when the problem size is small (TSP-30)

- The quality of the solution found by metaheuristic algorithms is better than that by hill climbers for most problems
- A hill climber should be the choice if the problem is convex
- The results reported here are obtained without enough parameter tuning (more careful investigation is needed)

Results of Numerical Optimization

	Convex		Griewank		Ackley	
Steepest Ascent	0.0 0.0	774,692	0.260 0.108	67,144	17.832 14.029	12,182
First Choice	0.0 0.0	274,521	0.254 0.064	38,824	18.214 14.108	14,377
Stochastic	0.0 0.0	2,088,171	0.218 0.096	387,008	18.879 16.729	141,156
Gradient Descent	**0.0** **0.0**	199,935	0.216 0.118	856,635	17.447 8.101	5,234
Simulated Annealing	0.0 0.0	500,000 63,565	0.367 0.145	500,000 18,252	19.319 18.940	500,000 5,541
GA	3.920 0.766	500,000 227,140	**0.036** **0.015**	500,000 220,390	**0.214** **0.141**	500,000 173,900

Results of Combinatorial Optimization (TSP)

	TSP-30		TSP-50		TSP-100	
Steepest Ascent	593 525	6,846	782 678	20,211	1,223 1,145	88,254
First Choice	424 408	30,154	578 561	57,207	903 869	131,450
Stochastic	422 408	670,952	**570** 561	2,395,138	872 840	10,903,041
Nearest Neighbor	509 455		694 646		958 918	
Simulated Annealing	**412** **408**	500,000 28,831	577 559	500,000 43,962	**829** **804**	500,000 69,084
GA	658 635	500,000 245,940	584 **558**	500,000 226,840	854 822	 422,650