
BioInspired Computing

Natural Computing Homework

Stephanie Athow

Paul Blasi

March 18, 2015

Contents

Title	i
Contents	iii
List of Figures	v
1 Evolutionary Algorithms - Text Chapter 3	1
1.1 Problem 1	1
1.2 Problem 2	3
1.2.1 Summary	3
1.2.2 Implementation	3
1.2.3 Analysis	3
1.3 Problem 7	5
2 Artificial Neural Networks - Text Chapter 4	9
2.1 Problem 17	9
3 Swarms - Text Chapter 5	11
3.1 Problem 1	11
3.1.1 Summary	11
3.1.2 Implementation	11
3.1.3 Analysis	12
3.1.4 S-ACO Pseudocode	12
3.2 Problem 8	12
3.2.1 Summary	12
3.2.2 Implementation	13
3.2.3 Analysis	13
4 Immunocomputing - Text Chapter 6	15
4.1 Problem 1	15
4.1.1 Summary	15
4.1.2 Implementation	15
4.1.3 Analysis	16
A Supporting Materials	17
A.1 Problem 7 results	17
B Code	23

List of Figures

1.1	Sample Hill Climb Algorithms Answers	2
3.1	Sample Particle Swarm Answers	13

Evolutionary Algorithms - Text Chapter 3

1.1 Problem 1

Implement the various hill-climbing procedures and the simulated annealing algorithm to solve the problem exemplified in Equation 1.1. Use a real-valued representation scheme for the candidate solutions (variable x).

By comparing the performance of the algorithms, what can you conclude?

For the simple hill-climbing try different initial configurations as attempts at finding the global optimum. Was this algorithm successful?

Discuss the sensitivity of all the algorithms in relation to their input parameters.

$$g(x) = 2^{-2((x-0.1)/0.9)^2} * \sin(5\pi x)^6 \quad (1.1)$$

Simple Hill Climb:

In the simple hill climb algorithm, a point is randomly selected in the search space, evaluated, slightly perturbed and re-evaluated until either a max iteration value or non-significant change happens. This algorithm easily finds local maximums or minimums but can miss the global extrema since there is no way to 'escape' a local extrema.

Iterated Hill Climb:

The iterated hill climb algorithm repeats the simple hill climb for a specified number of loops, with random starting points each time, and keeps track of the best solution. Since this simply reruns simple hill climb over and over with different starting points, local extremas can be 'escaped' or 'ignored' and the global extrema found.

Stochastic Hill Climb:

The stochastic hill climb is quite similar to the simple hill climb, with one change that makes a significant difference. When the point is perturbed, it is perturbed randomly and its acceptance as a new point is probabilistically determined with Equation 1.2 The value of T plays an important part in this equation. T determines the decay of the exponential function. In plain words, T determines how important the relative difference between the evaluation of x and x' is. A large T will make the search very similar to a random search and does not provide consistent results. When T is small, local extremas can be escaped and global extremas found with fairly consistent results.

Simulated Annealing:

The simulated annealing algorithm is very similar in concept to the stochastic hill climbing method. The difference is in the T value, which instead of being set, starts at a large value and becomes much smaller over the algorithm's execution. This allows the algorithm to start out as nearly random and lets it more fully cover the search space. As the algorithm continues, the probability of it jumping out of the current extrema gets smaller. This combination gets the algorithm to completely cover the search space and then concentrate itself on the global extrema instead of the local ones.

$$P = 1/(1 + \exp[(eval(x) - eval(x'))/T]) \quad (1.2)$$

A few sample runs of the various hill climb algorithms can be found in Figure 1.1.

```

C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X = 0.897613925322

Iterated Hill Climb
X = 0.099692725589

Stochastic Hill Climb
X = 0.0996053805818

C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X = 0.721759360058

Iterated Hill Climb
X = 0.10007646174

Stochastic Hill Climb
X = 0.101296488508

C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X = 0.901745241427

Iterated Hill Climb
X = 0.100168148946

Stochastic Hill Climb
X = 0.103018083057

C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X = 0.296639976009

Iterated Hill Climb
X = 0.100045357411

Stochastic Hill Climb
X = 0.0973894547446

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\1955241\Source\Repos\NatComp_HW2\BioInspiredHW\code>python SimulatedAnnealing.py
Iteration 10: x = 0.529855051421 ! Energy = 0.367247816108
Iteration 20: x = 0.698441889302 ! Energy = 0.540785697804
Iteration 30: x = 0.698441889302 ! Energy = 0.540785697804
Iteration 40: x = 0.314702209601 ! Energy = 0.786361213512
Iteration 50: x = 0.314702209601 ! Energy = 0.786361213512
Iteration 60: x = 0.299957962561 ! Energy = 0.933857414543
Iteration 70: x = 0.100025295956 ! Energy = 0.99999525249
Iteration 80: x = 0.100025295956 ! Energy = 0.99999525249
Iteration 90: x = 0.100025295956 ! Energy = 0.99999525249
Iteration 100: x = 0.100025295956 ! Energy = 0.99999525249

C:\Users\1955241\Source\Repos\NatComp_HW2\BioInspiredHW\code>python SimulatedAnnealing.py
Iteration 10: x = 0.603018248604 ! Energy = 7.34855380496e-09
Iteration 20: x = 0.505726640461 ! Energy = 0.736356614028
Iteration 30: x = 0.496364275993 ! Energy = 0.758418890259
Iteration 40: x = 0.08921568461 ! Energy = 0.916949695127
Iteration 50: x = 0.08921568461 ! Energy = 0.916949695127
Iteration 60: x = 0.08921568461 ! Energy = 0.916949695127
Iteration 70: x = 0.08921568461 ! Energy = 0.916949695127
Iteration 80: x = 0.08921568461 ! Energy = 0.916949695127
Iteration 90: x = 0.299183001223 ! Energy = 0.933891592797
Iteration 100: x = 0.299183001223 ! Energy = 0.933891592797

C:\Users\1955241\Source\Repos\NatComp_HW2\BioInspiredHW\code>python SimulatedAnnealing.py
Iteration 10: x = 0.551003544024 ! Energy = 0.0795343724354
Iteration 20: x = 0.506213324101 ! Energy = 0.732691063584
Iteration 30: x = 0.506213324101 ! Energy = 0.732691063584
Iteration 40: x = 0.304868040025 ! Energy = 0.914488053521
Iteration 50: x = 0.01008083665 ! Energy = 0.9924627741
Iteration 60: x = 0.0993166669724 ! Energy = 0.999653612744
Iteration 70: x = 0.0993166669724 ! Energy = 0.999653612744
Iteration 80: x = 0.0993166669724 ! Energy = 0.999653612744
Iteration 90: x = 0.0993166669724 ! Energy = 0.999653612744
Iteration 100: x = 0.0993166669724 ! Energy = 0.999653612744

C:\Users\1955241\Source\Repos\NatComp_HW2\BioInspiredHW\code>python SimulatedAnnealing.py
Iteration 10: x = 0.368324317636 ! Energy = 0.0104506196114
Iteration 20: x = 0.297124763537 ! Energy = 0.929948339201
Iteration 30: x = 0.297124763537 ! Energy = 0.929948339201
Iteration 40: x = 0.297124763537 ! Energy = 0.929948339201
Iteration 50: x = 0.297124763537 ! Energy = 0.929948339201
Iteration 60: x = 0.0996970771841 ! Energy = 0.999931920742
Iteration 70: x = 0.0996970771841 ! Energy = 0.999931920742
Iteration 80: x = 0.0996970771841 ! Energy = 0.999931920742
Iteration 90: x = 0.0996970771841 ! Energy = 0.999931920742
Iteration 100: x = 0.0996970771841 ! Energy = 0.999931920742

```

Figure 1.1: Sample Hill Climb Algorithms Answers

1.2 Problem 2

Implement and apply the hill-climbing, simulated annealing, and genetic algorithms to maximize function $g(x)$ used in the previous exercise assuming a bitstring representation.

Tip: The perturbation to be introduced in the candidate solutions for the hill-climbing and simulated annealing algorithms may be implemented similarly to the point mutation in genetic algorithms. Note that in this case, no concern is required about the domain of x , because the binary representation already accounts for it.

Discuss the performance of the algorithms and assess their sensitivity in relation to the input parameters.

1.2.1 Summary

Expanding on the evaluations in the previous problem we add a simple genetic algorithm to the mix. As mentioned, many of the solutions above can find themselves caught in a local extrema for many optimization problems. The genetic algorithm (granted a large enough initial population) avoids this problem by sampling a larger area of the search space.

1.2.2 Implementation

The code can be found in Listing B.1.

Encoding

For the gene encoding, we used an unsigned 16 bit integer. This gave us an easy way to store 16 bits. Since we know the domain of the problem is restricted to $[0, 1]$ decoding the genotype into a phenotype was done by dividing the integer value of the bit string by the maximum value of a 16 bit integer (65535). Because the values are stored as integers, we can use numpy's `random_integers` function to initialize the population.

Crossover & Mutation

Crossover and mutation were handled by binary operations. Binary masks were stored in lists for each operator and a random index into those lists was generated to pick the mask used.

For mutation the individual was XOR'd with the mask to flip the proper bit. For crossover one parent was AND'd with the chosen mask and the other was AND'd with that mask's negation. These values were then OR'd together to make the child individual.

Selection

Since our fitness values were simply the y values of the function and maximizing the function was the goal, a simple roulette selection method was chosen.

To avoid having to mess with the list of fitness values, an unused index list is created, and this is what the indexes are chosen from. Iterating through the unused list we add the value at that list to a variable until it is greater than or equal to a random value between 0 and the summation.

Between choosing each index the value at the chosen index is subtracted from the sum to avoid having to sum the unused values again. Unfortunately, since this uses floating point calculations it occasionally didn't reach the sum before the algorithm ran out of indexes. We fixed this problem by defaulting to the last index if the sum hasn't been reached by that point.

1.2.3 Analysis

Through multiple runs with different parameters, it was found that a genetic algorithm could solve this problem consistently with an initial population as low as 10 individuals in 250 iterations. The fastest

consistent solution came from runs with around 100 individuals and 100 iterations. If more iterations were run it was able to get more precise, giving an answer of .0999923704891 for a max of .999999956813. Taking into account that the encoding isn't perfect this probably means we have the global max at .1 with a value of 1.

In comparison with the other methods summarized in the previous problem the genetic algorithm is rather slow. But it has a much wider range of effective input parameters and when compared to a simple hill climbing algorithm gets the right solution much more often.

1.3 Problem 7

Determine, using genetic programming (GP), the computer program (S-expression) that produces exactly the outputs presented in Table 3.3 for each value of x . The following hypotheses are given:

- Use only functions with two arguments (binary trees).
- Largest depth allowed for each tree: 4
- Function set: $F = +, *$
- Terminal set: $T = 0, 1, 2, 3, 4, 5, x$

x	Program output
-10	153
-9	120
-8	91
-7	66
-6	45
-5	28
-4	15
-3	6
-2	1
-1	0
0	3
1	10
2	21
3	36
4	55
5	78
6	105
7	136
8	171
9	210
10	253

Summary

For this problem we originally intended to use BNF encoding and the related crossover and mutation operators with it. We were not able to get BNF encoding to work well as we were unsure of how to validate a string of integers as being a valid string. We then moved to an adapted simple genetic algorithm that worked on a prefix format equation. This encoding made validating the string and ensuring that the resulting tree was limited to a depth of four much easier.

Encoding

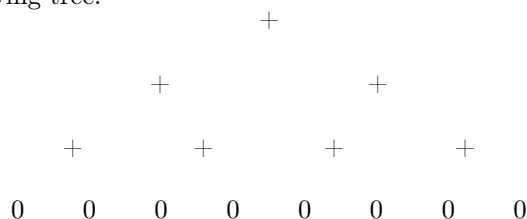
As stated above the encoding is as a prefix formatted equation. This format is easy to do crossover with (when we set the format of the equation in stone), but mutation has to check for two different cases of mutation. One for operators and one for terminals. The operators and terminals are given from the problem and explicitly stated below.

Operators	Terminals
+ *	0
	1
	2
	3
	4
	5
	x

To simplify the encoding process, we started with a default encoding and then used the mutation operator to randomize the initial population. The default encoding can be seen below.

+ + + 0 0 + 0 0 + + 0 0 + 0 0

This evaluates to the following tree:



As you can see, this is not a particularly useful individual, so we copy it and randomize it through mutation to sample the search space.

Crossover & Mutation

Crossover was even simpler in this problem than in the previous. Our prefix equation was stored as a list of strings, so we split the mother and father lists at a random crossover point and recombined them into a child.

Mutation was more complicated in this problem but only marginally so. The complication was that there are two parts to the individuals using our encoding. There were the operators and the terminals. Since the locations of these are fixed, we stored the locations of each in a separate list, and if our randomly selected mutation index is in the operator index list we replace the value with a random operator. If the mutation index is in the terminals index list we replace the value with a random terminal.

Selection

The roulette selection operator from the previous problem was reused. To make it work properly, the evaluation function had to be modified significantly. Our first instinct was to add up the difference between the expected and found values. This was easy to implement so we went with it, but this evaluation function does the exact opposite of what we want. It creates high values for less fit individuals. We solved this by finding the maximum value and then replacing every individual's fitness with $(max - fitness)/max$. This normalized the values between 0 and 1 and made them be the proper fitness relative to each other.

Analysis

The hardest part of this problem was to choose a good representation for the individuals. The prefix notation worked well and forcing a format that expanded to a full depth four tree simplified a lot of the process.

When using a population of 250 and up to 1000 iterations (up to because we can short circuit the process if we find a perfect solution) we calculated a %100 match 100 times out of 100. The full list of these can be found in Appendix A.1.

Because we didn't want to fully analyze 100 different prefix notation equations, we randomly picked 10 to evaluate. These are listed below.

Solution	Prefix	Simplified
52	$+ + * x x * x x + * x 4 + 3 x$	$2x^2 + 5x + 3$
81	$+ * + 0 x + x x + + 2 1 * 5 x$	$2x^2 + 5x + 3$
49	$+ + + 3 x * x x * + x 0 + x 4$	$2x^2 + 5x + 3$
93	$* + + x 0 + 0 1 + + 2 1 * 2 x$	$2x^2 + 5x + 3$
47	$+ + * x 5 + 3 0 * + 0 2 * x x$	$2x^2 + 5x + 3$
26	$* + + x 2 + x 1 + + x 0 + 1 0$	$2x^2 + 5x + 3$
62	$+ + * x x + 3 x + * x 4 * x x$	$2x^2 + 5x + 3$
68	$* + + x x + 0 3 + * 2 0 + 1 x$	$2x^2 + 5x + 3$
42	$* + * 0 3 + x 1 + + 3 x + 0 x$	$2x^2 + 5x + 3$
29	$+ * + 2 x + x x + + x 3 + 0 0$	$2x^2 + 5x + 3$

Summarizing these, it looks like the equation is

$$2x^2 + 5x + 3 \tag{1.3}$$

Artificial Neural Networks - Text Chapter 4

2.1 Problem 17

Apply the MLP network trained with the backpropagation learning algorithm to solve the character recognition task of Section 4.4.2.

Determine a suitable network architecture and then test the resultant network sensitivity to noise the the test data. Test different noise levels, from 5% to 50%

Since this problem was extra credit we devoted more time to other problems with the given extension.

Swarms - Text Chapter 5

3.1 Problem 1

Write a pseudocode for the simple ACO (S-ACO) algorithm considering pheromone evaporation, implement it computationally, and apply it to solve the TSP instance presented in Section 3.10.4. Discuss the results obtained.

Remove the pheromone evaporation term (Equation 3.1), apply the algorithm to the same problem, and discuss the results obtained.

$$\tau_{ij}(t) \leftarrow (1 - \rho)\tau_{ij}(t) + \Delta\tau \quad (3.1)$$

3.1.1 Summary

The Traveling Salesman Problem is a popular problem in the field of computer science. The idea is there is a list of cities a salesman wants to visit, for the cheapest cost. In this problem, ant colony optimization is used to find a low-cost solution.

3.1.2 Implementation

Code for this may be found in Listing B.3.

The pseudocode for the simple ant colony optimization is presented in subsection 3.1.4. This pseudocode has gone through a few revisions as code was written and moments of “That’s not going to work.” occurred. Currently, the code is written to read in cities locations from a file and create edges to form a complete graph with associated distances, pheromone, and visibilities. Next, a solution for each ant is built. A node is randomly selected for the starting point, a probability list (for moving to the next unvisited node) is constructed using Equation 3.2, a random value is generated and checked against the probability list to see which node is next moved to. The selected node is then removed from the available nodes list, added to the visited list, and pheromone added to the edge. This repeats for each node until all nodes have been traversed.

$$p_{ij}^k(t) = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in J_i^k} [\tau_{il}(t)]^\alpha [\eta_{il}]^\beta} & \text{if } j \in J_i^k \\ 0 & \text{otherwise} \end{cases} \quad (3.2)$$

Once the visited list for each ant has been built, the fitness of each path will be evaluated. To determine the fitness, it will simply be the total length of all edges. The most fit path will be the shortest path. Then, Equation 3.3 is applied to all edges to simulate the ‘evaporation’ of pheromones.

$$\tau_{ij}(t) \leftarrow \tau_{ij}(t) + \Delta\tau \quad (3.3)$$

3.1.3 Analysis

There's a bug in the bug code. A struct to contain edge information such as start node, end node, pheromone level, distance and visibility was created. Then a 2D array of those structs was created to hold all possible edges. The 2D array is filled such that no edge is repeatedly stored, so half of the array is filled. Pheromone is initialized to one for all edges.

During debugging of the move function, when accessing the pheromone information, occasionally the pheromone data would be 0 instead of 1. This could be an indexing issue, but on 3 hours of sleep, and checking the logic against several people, Stephanie can't figure out what's wrong. One would think that she would have learned from McGough's robotics class, his homeworks should really be started more than one week before it's due.

3.1.4 S-ACO Pseudocode

```
SACO( max_it, ants, edges )
{
    t = 0
    while ( t < max_it )
    {
        // for each ant
        // place ant on randomly selected node

        for( i = 0; ants; i++)
        {
            // calculate probability of moving to each untraveled node
            // use probabilistic move rule to determine next node
            // mark new node as traveled
            // add node to traveled path
        }

        // evaluate cost of each solution
        if ( solution < best )
            best = solution

        //update pheromone trails
        t++
    }
}
```

3.2 Problem 8

Apply the PS algorithm described in Section 5.4.1 to the maximization problem of Example 3.3.3. Compare the relative performance of the PS algorithm with that obtained using a standard genetic algorithm.

3.2.1 Summary

Expanding upon the evolutionary algorithms to solve the maximization problem of Example 3.3.3 in the book, a particle swarm approach is implemented and compared against the evolutionary algorithms.

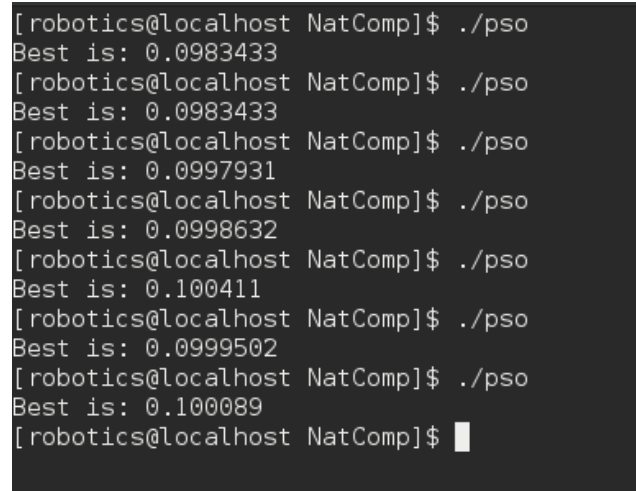
3.2.2 Implementation

Random points along the x-axis between zero and one are initialized as the population. Each individual is then assigned a random initial velocity between zero and one. Next, each individual's fitness is then evaluated and the best fit individual is tracked. Then a loop is entered for a specified number of time steps. During each time step, each individual's velocity is slightly perturbed, the fitness re-evaluated, and new best fit positions are tracked. The code can be found in Listing B.2.

3.2.3 Analysis

The trials for this implementation were run with a population size of twenty and 500 time steps. With these conditions, the particle swarm successfully found the global max within a small tolerance each time it was run, see Figure 3.1 for some sample runs. Compared to the simple hill climbing algorithm, particle swarm will always find the global extrema, whereas the simple hill climber will not. So, particle swarm is better than the simple hill climber. The iterated and stochastic hill climbers consistently found the global extrema, so particle swarm is on par with them. The same can be said for the genetic algorithm.

Comments on run times can't be accurately made because python was used for some algorithms and c++ for the others.

A terminal window with a dark background and light-colored text. It shows ten consecutive runs of a program named 'pso'. Each run starts with a prompt '[robotics@localhost NatComp]\$./pso' followed by the output 'Best is: ' and a numerical value. The values are: 0.0983433, 0.0983433, 0.0997931, 0.0998632, 0.100411, 0.0999502, 0.100089, and the last line is cut off with a block character. The prompt for the last run is also cut off.

```
[robotics@localhost NatComp]$ ./pso
Best is: 0.0983433
[robotics@localhost NatComp]$ ./pso
Best is: 0.0983433
[robotics@localhost NatComp]$ ./pso
Best is: 0.0997931
[robotics@localhost NatComp]$ ./pso
Best is: 0.0998632
[robotics@localhost NatComp]$ ./pso
Best is: 0.100411
[robotics@localhost NatComp]$ ./pso
Best is: 0.0999502
[robotics@localhost NatComp]$ ./pso
Best is: 0.100089
[robotics@localhost NatComp]$ █
```

Figure 3.1: Sample Particle Swarm Answers

Immunocomputing - Text Chapter 6

4.1 Problem 1

Use a bone marrow algorithm to define genes for gene libraries to be used to generate the initial population of a genetic algorithm to solve the TSP problem presented in Chapter 3 and Chapter 5 (Figure 6.24). Assume the following structure for the gene libraries:

Gene length $L_g = 4$, number of libraries $n = 8$, and library length (number of genes in each library) $L_l = 4$. As one gene for each library will be selected, the total chromosome length is $L = L_g \times n = 4 \times 8 = 32$, that corresponds to the number of cities in a tour.

Each gene will be defined as a sequence of four cities known to be part of an optimal route. Since many of the chromosomes produced will contain repeated values, a repair function must be created.

Implement this bone marrow model to define an initial population of chromosomes to be used in an evolutionary algorithm to solve the TSP problem illustrated. Compare the performance of the algorithm with this type of initialization procedure and with the random initialization used in Project 1, Section 3.10.4

4.1.1 Summary

Since we did not do Project 1 for this homework assignment, we won't be comparing the performance of the populations. This algorithm is pretty simple, so we packaged it into a reusable class for future work. The code can be found in Listing B.5.

4.1.2 Implementation

As stated above, we packaged the process into a generic class to allow it to be used for various problems.

Libraries

We decided to let there be two methods of adding a library. You can add a complete library, or create a library with a gene generator function. In some initial thoughts, a mutator function could also be passed in, but this was out of the scope of the function and not necessary for the problem.

Chromosomes

With our class you can create single or sets of chromosomes. The chromosomes are run through a repair function that is passed into the constructor of the class before being returned or added to the return list. The method of making the chromosomes is pretty straight forward. Start with a random element from the first library and add random genes from the rest of the libraries in the order added until the chromosome is complete.

4.1.3 Analysis

This seems to be a quick way to generate more complex encodings than using a purely random method. It would have been a nice addition to Chapter 3 Problem 7, as the prefix notation was made of very specific parts, but they were different from each other and had to go in order. It may be used in that problem for future analysis

In our testing two different generators were used, one with random strings for genes, and one with a subset of the optimal path (seen in Figure 6.24 in the text). Running the python file generates and prints both of these sets to the screen.

A

Supporting Materials

A.1 Problem 7 results

0 Eval: 1.0

+ + * x x * x x + * 5 x + 3 0

1 Eval: 1.0

+ + * x x + x 3 * + 4 x + 0 x

2 Eval: 1.0

+ * + x x + x 2 + + 3 0 * x 1

3 Eval: 1.0

+ * + 2 x + x x + + 0 3 + 0 x

4 Eval: 1.0

+ * + 5 x + x 0 + + 3 0 * x x

5 Eval: 1.0

* + * 1 3 + x x + + 0 1 + 0 x

6 Eval: 1.0

+ + * x 3 + 0 3 * + x x + x 1

7 Eval: 1.0

+ * * x 2 + x 0 + * 5 x + 2 1

8 Eval: 1.0

+ + + x 0 + 1 2 * + x x + 2 x

9 Eval: 1.0

+ * + x 2 + x x + + 3 x * 0 3

10 Eval: 1.0

+ + + 3 x * x x * + x 4 + x 0

11 Eval: 1.0

+ + * x x * 5 x + + 3 0 * x x

12 Eval: 1.0

+ * + 2 x + x x + + x 3 * 0 5

13 Eval: 1.0

+ + * x x + 1 2 * + x 5 + 0 x

14 Eval: 1.0

* + + 0 1 + 0 x + + x 0 + 3 x

15 Eval: 1.0

+ * + x 3 + 1 x + * x 1 * x x

16 Eval: 1.0

+ + + 3 0 * 1 x * * 2 x + x 2

17 Eval: 1.0

+ + * x x * x 5 + * x x + 1 2

18 Eval: 1.0

* + + 2 x + x 1 + + 1 x + 0 0

19 Eval: 1.0

+ * + 1 x + 3 0 * * 2 x + x 1

20 Eval: 1.0

+ * + x x + x 2 + + 2 x + 0 1

21 Eval: 1.0

* + + x 0 + x 3 + + 0 1 + x 0

22 Eval: 1.0

+ * + x x + x 2 + + 3 x + 0 0

23 Eval: 1.0

+ + + 2 1 + x 0 * + x 2 + x x

24 Eval: 1.0

+ * + 3 x + 1 x * + x 0 + x 1

25 Eval: 1.0

* + + 3 0 + x x + + 1 x + 0 0

26 Eval: 1.0

* + + x 2 + x 1 + + x 0 + 1 0

27 Eval: 1.0

+ + * x x + 3 x + * x x * 4 x

28 Eval: 1.0

+ * + 0 x + x 1 * + x 3 + 1 x

29 Eval: 1.0

+ * + 2 x + x x + + x 3 + 0 0

30 Eval: 1.0

* + + 0 1 + x 0 + + x 0 + 3 x

31 Eval: 1.0

+ + * 5 x * x x + * x x + 1 2

32 Eval: 1.0

+ * * x 2 + x 2 + + 2 0 + 1 x

33 Eval: 1.0

+ * + 2 x + x x + + 3 0 + x 0

34 Eval: 1.0

* + + x 3 + x 0 + + 0 0 + x 1

35 Eval: 1.0

* + + 1 0 + 0 x + + x x + 3 0

36 Eval: 1.0

* + + 1 0 + x 0 + + x 3 + x 0

37 Eval: 1.0

+ * + x 2 + x x + + 2 x + 0 1

38 Eval: 1.0

+ * + 1 x + x 3 + + x 0 * x x

39 Eval: 1.0

+ * + 2 x + x x + + 2 0 + 1 x

40 Eval: 1.0

* + + 0 x + 1 0 + + x 2 + 1 x

41 Eval: 1.0

+ * + x 1 + x 3 + + x 0 * x x

42 Eval: 1.0

* + * 0 3 + x 1 + + 3 x + 0 x

43 Eval: 1.0

* + + x 3 + x 0 + + x 0 + 1 0

44 Eval: 1.0

+ + * x 4 * x x + + 3 x * x x

45 Eval: 1.0

+ * * 2 x + 2 x + + 0 1 + 2 x

46 Eval: 1.0

* + + x x * 3 1 + + 1 x * 0 x

47 Eval: 1.0

+ + * x 5 + 3 0 * + 0 2 * x x

48 Eval: 1.0

* + + 3 x + x 0 + * 0 4 + x 1

49 Eval: 1.0

+ + + 3 x * x x * + x 0 + x 4

50 Eval: 1.0

+ * + 1 x + x x * + 3 0 + 1 x

51 Eval: 1.0

+ * + x x + 2 x + + 0 3 + x 0

52 Eval: 1.0

+ + * x x * x x + * x 4 + 3 x

53 Eval: 1.0

* + + 3 x + x 0 + + 0 x + 1 0

54 Eval: 1.0

+ + * x x + 0 x * + x 3 + 1 x

55 Eval: 1.0

+ * + x 2 + x x + + x 0 + 1 2

56 Eval: 1.0

+ + * x x * x x + * 5 x + 3 0

57 Eval: 1.0

+ + + 1 0 + 2 x * + x x + 2 x

58 Eval: 1.0

+ * * 2 x + x 2 + + x 3 * 0 0

59 Eval: 1.0

+ * + 3 0 + x 1 * * x 2 + 1 x

60 Eval: 1.0

* + + 1 0 + x 0 + + x 0 + x 3

61 Eval: 1.0

+ + * 5 x + 1 2 + * x x * x x

62 Eval: 1.0

+ + * x x + 3 x + * x 4 * x x

63 Eval: 1.0

+ * + x x + 2 x + * 0 0 + 3 x

64 Eval: 1.0

+ * + 0 x + x 5 + + 2 1 * x x

65 Eval: 1.0

* + + 0 x + x 3 + + x 0 + 1 0

66 Eval: 1.0

* + * 0 0 + x 1 + + 3 0 + x x

67 Eval: 1.0

* + + 0 3 + x x + + x 1 * 5 0

68 Eval: 1.0

* + + x x + 0 3 + * 2 0 + 1 x

69 Eval: 1.0

+ + * x x + 3 0 + * x 5 * x x

70 Eval: 1.0

+ * + 1 x + x 3 + * x x + 0 x

71 Eval: 1.0

+ * + 2 x + x x + + 3 x * 4 0

72 Eval: 1.0

* + + 0 x + x 3 + + 1 x + 0 0

73 Eval: 1.0

+ + * x 5 * 3 1 + * x x * x x

74 Eval: 1.0

* + + x 3 + 0 x + + 0 x + 1 0

75 Eval: 1.0

+ * + x x + 2 x + * 0 3 + x 3

76 Eval: 1.0

+ * + x 2 + x x + + 0 3 + x 0

77 Eval: 1.0

+ + * 0 3 + 3 x * + x x + x 2

78 Eval: 1.0

+ + + x 0 + 3 0 * + 2 x + x x

79 Eval: 1.0

+ + + x 2 + 1 0 * + x x + x 2

80 Eval: 1.0

+ * + x x + x 2 + + x 3 * 5 0

81 Eval: 1.0

+ * + 0 x + x x + + 2 1 * 5 x

82 Eval: 1.0

+ * + 5 x + 0 x + * x x + 3 0

83 Eval: 1.0

+ * + 0 x + x 5 + * 1 3 * x x

84 Eval: 1.0

+ * + 2 x + x x + * x 0 + 3 x

85 Eval: 1.0

+ * + 2 x + x x + + 0 3 + x 0

86 Eval: 1.0

+ * + 0 1 + 3 x * + x x + x 2

87 Eval: 1.0

+ + + 3 x + 0 0 * + 2 x * x 2

88 Eval: 1.0

* + + 0 3 * x 2 + + x 1 + 0 0

89 Eval: 1.0

+ + * x 5 + 3 0 * * 2 x + 0 x

90 Eval: 1.0

* + + x x + 0 3 + + 1 x * 3 0

91 Eval: 1.0

+ * + x 2 + x x + + x 0 + 3 0

92 Eval: 1.0

+ * + x 3 + 1 0 * + x 2 * x 2

93 Eval: 1.0

* + + x 0 + 0 1 + + 2 1 * 2 x

94 Eval: 1.0

+ * + 2 x + x x + + 1 x + 0 2

95 Eval: 1.0

+ * + 2 0 * x x + * x 5 + 1 2

96 Eval: 1.0

+ + * x x + x 3 * + x 4 + x 0

97 Eval: 1.0

+ * + x x + x 2 + + 0 0 + x 3

98 Eval: 1.0

+ * + x x + x 2 + + 2 x + 0 1

99 Eval: 1.0

* + + x 3 + 0 x + * 0 1 + 1 x

Code

```
from __future__ import print_function
import numpy as np
import math
import random

class EA:
    bit_masks = ( 0b000000000000000001,
                  0b000000000000000010,
                  0b000000000000000100,
                  0b00000000000001000,
                  0b00000000000010000,
                  0b00000000000100000,
                  0b00000000001000000,
                  0b00000000010000000,
                  0b00000000100000000,
                  0b00000001000000000,
                  0b00000010000000000,
                  0b00000100000000000,
                  0b00001000000000000,
                  0b00010000000000000,
                  0b00100000000000000,
                  0b01000000000000000,
                  0b10000000000000000 )

    reproduction_masks = ( 0b111111111111111100,
                           0b111111111111111100,
                           0b111111111111111000,
                           0b111111111111110000,
                           0b111111111111100000,
                           0b111111111111000000,
                           0b111111111110000000,
                           0b111111111100000000,
                           0b111111111000000000,
                           0b111111110000000000,
                           0b111111100000000000,
                           0b111111000000000000,
                           0b111110000000000000,
                           0b111100000000000000,
                           0b111000000000000000,
                           0b110000000000000000 )

    binary_format = '{:#018b}'
```

```

def __init__(self, population_size, mutation_rate):
    self.population_size = population_size
    self.child_population_size = self.population_size // 2
    self.mutation_rate = mutation_rate
    self.population = np.random.random_integers(0, 65535, self.population_size).as
    self.population_eval = np.zeros(dtype=np.float, shape=(self.population_size))
    self.__next_population = np.zeros(dtype=np.uint16, shape=(self.population_size
+ self.child_population_size))
    self.__next_population_eval = np.zeros(dtype=np.float, shape=(self.population_
self.__child_population = np.zeros(dtype=np.uint16, shape=(self.child_populati
self.__child_population_eval = np.zeros(dtype=np.float, shape=(self.child_popu
self.__child_pop_indices = range(self.child_population_size)
self.__pop_indices = range(self.population_size)

    self.evaluate(self.population, self.population_eval);

def print_best(self):
    index = 0
    max = 0
    for i in self.__pop_indices:
        if self.population_eval[i] > max:
            max = self.population_eval[i]
            index = i
    print(self.binary_format.format(self.population[index]), "\nX:", np.float32(se

def evaluate(self, pop, eval):
    for i in range(pop.size):
        x = np.float32(pop[i]) / 65535
        eval[i] = (2 ** (-2 * (((x - 0.1)/.9) * ((x - 0.1)/.9))) * math.sin(5 * ma

def select_indices(self, count, eval):
    unused = range(eval.size)
    indices = []
    i = 0
    sum = np.sum(eval)

    while i < count:
        i = i+1
        rand = np.random.uniform(0, sum)
        index = 0
        curr = eval[unused[index]]

        while curr < rand and index < len(unused) - 1:
            index = index + 1
            curr = curr + eval[unused[index]]

        sum = sum - eval[unused[index]]
        indices.append(unused.pop(index))

    return indices

def create_children(self, indices):
    for i in self.__child_pop_indices:

```

```

        child = self.combine(self.population[indices[2 * i]], self.population[indices[2 * i + 1]])
        if np.random.uniform() < self.mutation_rate:
            child = self.mutate(child)

        self.__child_population[i] = child

    self.evaluate(self.__child_population, self.__child_population_eval)

    for i in self.__child_pop_indices:
        self.__next_population[self.population_size + i] = self.__child_population[i]
        self.__next_population_eval[self.population_size + i] = self.__child_population_eval[i]

    def combine(self, mom, dad):
        mask = self.reproduction_masks[random.randint(0, len(self.reproduction_masks))]
        return (mom & mask) | (dad & ~mask)

    def mutate(self, child):
        mask = self.bit_masks[random.randint(0, len(self.bit_masks) - 1)]
        return child ^ mask

    def run(self, iterations, printEvery):
        i = 0
        while i < iterations:
            i = i + 1

            for j in self.__pop_indices:
                self.__next_population[j] = self.population[j]
                self.__next_population_eval[j] = self.population_eval[j]

            self.create_children(self.select_indices(self.population_size, self.population_eval, self.__pop_indices))

            indices = self.select_indices(self.population_size, self.__next_population_eval, self.__pop_indices)

            for j in self.__pop_indices:
                self.population[j] = self.__next_population[indices[j]]
                self.population_eval[j] = self.__next_population_eval[indices[j]]

            if i != iterations and i % printEvery == 0:
                self.print_best();

        print("\nSOLUTION\n-----")
        self.print_best();

if __name__ == '__main__':
    test = EA(250, .1)
    test.run(250, 1)

```

Listing B.2: pso.cpp

```
// Author: Stephanie Athow
// Date: 16 March 2015
// Professor: Dr. McGough
//
// Problem: "Fundamentals of Natural Computing", pg 260, #8
// Apply ths PS algorithm described in Section 5.4.1 to the maximization
// problem of Example 3.3.3.
//
// Code Credit: Dr. Jeff McGough,
// Lecture Slides - Ch5: function prototypes and main

// Includes
#include <iostream>
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <math.h>

using namespace std;

// Defines
#define POP 20
#define STEPS 200
#define _USE_MATH_DEFINES

// Function Prototypes
void initswarm( double swarm[] );
void initvel( double vel[] );
void init( double swarm[], double bestind[], double fitbest[] );
void printswarm( double swarm[], double fitness[] );
void printvel( double vel[] );
double fitswarm( double swarm[], double fitness[] );
double fit( double x );

// Functions

// Main - runs the helper functions
// Author: Dr. McGough
int main()
{
    int i, j = 0;
    double swarm[POP], vel[POP], bestind[POP], fitness[POP], fitbest[POP];
    double best, top, phi1, phi2;

    srand( time(NULL) );
    initswarm( swarm );
    best = fitswarm( swarm, fitness );
    top = fit( best );

    //cout << "x = " << best << endl;

    while( j++ < STEPS )
```



```

{
    // update positions and re-evaluate
    for( i = 0; i < POP; i++ )
    {
        // acceleration constants
        phi1 = rand()%200/1000.0 - 0.1;
        phi2 = rand()%200/1000.0 - 0.1;

        // new velocity
        vel[i] += phi1*(bestind[i] - swarm[i]) + phi2*(best - swarm[i]);

        // prevent invalid velocities
        if( (swarm[i] + vel[i] > 0.0) && (swarm[i]+vel[i] < 2.0) )
        {
            swarm[i] += vel[i];
            fitness[i] = fit( swarm[i] );
        }
    }

    // update best fit for each individual and best individual
    for( i = 0; i < POP; i++ )
    {
        if( fitness[i] > fitbest[i] )
        {
            bestind[i] = swarm[i];
            fitbest[i] = fitness[i];
        }

        if( fitness[i] > top )
        {
            best = swarm[i];
            top = fitness[i];
        }
    }
}

cout << "Best is: " << best << endl;

return 0;
}

// initialize particle positions
// Author: Stephanie Athow
void initswarm( double swarm[] )
{
    int i = 0;

    for( i = 0; i < POP; i++ )
    {
        swarm[i] = ( rand()/ (double)(RAND_MAX) );
        //cout << "pos = " << swarm[i] << endl;
    }
    return;
}

```

```
// initialize particle velocities
// Author: Stephanie Athow
void initvel( double vel[] )
{
    int i = 0;

    for( i = 0; i < POP; i++ )
    {
        vel[i] = (rand() / (double)(RAND_MAX) );
    }

    return;
}

// initialize particle positions, find best individual and best fit
void init( double swarm[], double bestind[], double fitbest[] )
{
    return;
}

//print particle fitnesses
void printswarm( double swarm[], double fitness[] )
{
    return;
}

// print particle velocities
void printvel( double vel[] )
{
    return;
}

// determine each particle's fitness
// Author: Stephanie Athow
double fitswarm( double swarm[], double fitness[] )
{
    int i = 0;
    double best_eval = 0.0;
    double best_x;

    for( i = 0; i < POP; i++ )
    {
        fitness[i] = fit( swarm[i] );
        //cout << "x: " << swarm[i] << " = " << fitness[i] << endl;

        if( (fitness[i]) > best_eval )
        {
            best_eval = fitness[i];
            best_x = swarm[i];
            //cout << "best " << endl;
        }
    }
}
```

```
    return best_x;
}

// Equation evaluation from example 3.3.3
//  $g(x) = 2^{(-2((x-0.1)/0.9)^2)} * (\sin(5\pi x))^6$ 
// Author: Stephanie Athow
double fit( double x )
{
    double exponent, fit;

    if( ( x > 0 ) && ( x < 1 ) )
    {
        exponent = -2 * pow( ( ( x - 0.1 ) / 0.9 ) , 2);
        fit = pow(2, exponent) * pow( (sin(5*M_PI*x)), 6 );
    }

    else
        fit = -10;

    return fit;
}
```

Listing B.3: tsp_aco.cpp

```

// Author: Stephanie Athow
// Date: 15 March 2015
// Professor: Dr. Jeff McGough
// Program: Traveling Salesman Problem using Ant Colony Optomization

#include <iostream>
#include <fstream>
#include <cmath>
#include <ctime>
#include <cstdlib>

using namespace std;

struct Edge
{
    int city1;
    int city2;
    float pheromone;      // amount of pheromone
    float distance;      // distance between cities
    float visibility;
    bool visit;          // if edge has been visited
};

// Function prototypes
float saco( int max_it, int ants, Edge edges[31][31] );
float path_eval( int paths[][31], Edge edges[31][31] );
float sum_pheromone( int curr, int city_traveled[], Edge edges[][31] );
void prob_list( int curr, int sum_p, float probabilities[31], int city_traveled[], Edg

// Functions
float saco( int max_it, int ants, Edge edges[31][31] )
{
    int paths[ants][31];
    int i, j, k, m, t = 0;
    int curr = 0;
    int city_traveled [31] = {0};
    int path_counter = 0;

    float sum_p = 0.0;
    float probability = 0.0;
    float probabilities[31];
    float path_try;
    float path_best = 10000.0;

    while( t < max_it )
    {
        // build a solution for each ant
        for( i = 0; i < 1; i++ )
        {

```

```

cout << "ant number: " << i << endl;

// place on random starting node
curr = rand()%32;
paths[i][0] = curr;
path_counter++;
city_traveled[curr] = 1;
cout << "rand curr: " << curr << endl;

while(path_counter < 2)
{
    //cout << "path count: " << path_counter << endl;

    // find sum of pheromones on remaining untraveled edges
    sum_p = 0;
    //sum_p = sum_pheromone( curr, city_traveled, edges );
    // array is not being filled correctly!

    //cout << "Sum pheromone: " << sum_p << endl;
    //cout << "curr: " << curr << endl;
    path_counter ++;

    //spoof sum_p for debugging other functions
    sum_p = 32 - path_counter;

    // generate probability list for current node
    prob_list( curr, sum_p, probabilities, city_traveled, edges );

    // cout << "probability list" << endl;

    // create move chance
    double move = ( (double)rand()/(RAND_MAX) ) ;

    for( int z = 0; z < 32; z++ )
    {
        if( probabilities[z] > move )
        {
            city_traveled[j] = 1;
            paths[i][path_counter] = j;
            path_counter++;
            curr = z;
            //cout << "moved to: " << z << endl;
            break;
        }
    }
    //cout << "moved to: " << z << endl;
}

// reset all information to zero for next ant
for( k = 0; k < 32; k++ )
    city_traveled[k] = 0;
path_counter = 0;

cout << "reset info" << endl;

```

```

    }

    // evaluate paths
    //path_try = path_eval( paths, edges );
    //if( path_try < path_best)
    //    path_best = path_try;
    // cout << "path eval" << endl;

    // time for next iteration!
    t++;
}

return path_best;
}

// evaluate path lengths
float path_eval( int paths[][31], Edge edges[31][31] )
{
    float path_best = 100000.0;
    float path_try = 0;
    int i, j, k, m = 0;
    int city1, city2;

    // find path lengths for each ant
    for( i = 0; i < 32; i++)
    {
        path_try = 0.0;

        for( j = 0; j < 31; j++)
        {
            city1 = paths[i][j];
            city2 = paths[i][j+1];
            path_try += edges[city1][city2].distance;
        }

        // compare path_try against best
        if( path_try < path_best)
            path_best = path_try;
    }

    return path_best;
}

// calculate pheromone sum for untraveled edges
float sum_pheromone( int curr, int city_traveled[], Edge edges[][31] )
{
    int i = 0;
    int j = 0;
    float sum = 0;

    // find sum of pheromones on remaining untraveled edges from curr node
    while( i < curr )

```

```

{
    if( city_traveled[i] == 0 )
    {
        cout << "traveled to: " << i << endl;
        sum += edges[i][curr-1-i].pheromone;
        cout << "sum: " << sum << endl;
        cout << "pheromone: " << edges[i][curr-1-i].pheromone << endl;
    }
    i++;
}

cout << endl << "i: " << i << endl;
cout << "p1: " << sum << endl;

while( i < 31)
{
    if( city_traveled[i] == 0 )
        sum += edges[curr+1][j].pheromone;
    i++;
    j++;
}

cout << "i2: " << i << endl;
cout << "p2: " << sum << endl;

return sum;
}

// generate probability that ant will move to node
void prob_list( int curr, int sum_p, float probabilities[31], int city_traveled[], Edg
{
    int i = 0;
    int j = 0;
    float alpha = 10.0;
    float beta = 0.0;
    float probability = 0;

    cout << "curr: " << curr << endl;

    while( i < curr )
    {
        if( city_traveled[i] == 0 )
        {
            probability += ( pow(edges[i][curr-1-i].pheromone, alpha ) *
                pow(edges[i][curr-1-i].visibility, beta)) /
                (sum_p * pow(edges[i][curr-1-i].visibility, beta ) );
            probabilities[i] = probability;
        }

        else if( city_traveled[i] == 1 )
            probabilities[i] = probabilities[i-1];
        i++;
        cout << "i: " << i << " prob: " << probability << endl;
    }
}

```

```

while( i < 32 )
{
    if( city_traveled[i] == 0 )
    {
        probability += ( pow(edges[curr+1][j].pheromone,alpha ) *
            pow(edges[curr+1][j].visibility,beta))/
            (sum_p * pow(edges[curr+1][j].visibility, beta ) );
        probabilities[i] = probability;
    }

    else if( city_traveled[i] == 1 )
        probabilities[i] = probabilities[i-1];
    i++;
    j++;
    cout << "i: " << i << " prob: " << probability << endl;
}

//cout << "end prob fun" << endl;
return;
}

int main()
{
    int x, y, index = 0;    // read in city information from text file
    int i, j;               // index counters
    int cities[31][2];      // holds city locations from file
    Edge edges[31][31];     // holds an array of Edges (struct)
    ifstream fin ("TSP.txt"); // file to read in city information
    float xdist, ydist;     // used to calc dist between cities
    float path_best;

    //cout << "after inits" << endl;

    // read in cities locations from file
    // NOTE: my file starts cities at 1, so it's index-1 to get correct
    // index number
    if ( fin.is_open() )
    {
        //cout << "file open" << endl;
        while( fin >> index )
        {
            //cout << "index is: " << index << endl;
            fin >> x >> y;
            cities[index - 1][0] = x;
            cities[index - 1][1] = y;
        }
    }

    // cout << "after if" << endl;

    // Fill edges array with all correct information
    for( i = 0; i < 32-1; i++ )
    {

```



```
for( j = i+1; j < 32; j++)
{
    edges[i][j].city1 = i;
    edges[i][j].city2 = j;
    // calc x and y distances for distance formula
    xdist = pow( ( cities[i][0] - cities[j][0] ), 2 );
    ydist = pow( ( cities[i][1] - cities[j][1] ), 2 );
    edges[i][j].distance = sqrt( xdist + ydist );
    edges[i][j].pheromone = 1.0;
    //edges[i][j].visit = false;
    edges[i][j].visibility = 1 / edges[i][j].distance;
}
}

// cout << "after fill edge array" << endl;

srand( time(NULL) );

// run simple aco
path_best = saco( 1, 32, edges );

cout << "Best distance is: " << path_best << endl;
// cout << "saco ran" << endl;

// close file!
fin.close();

return 1;
}
```

Listing B.4: nc_hw2_hill_climb.py

```
#####
# Project: Hill Climb Programs
# Author: Stephanie Athow
# Date: 11 February 2015
# Professor: Dr. McGough
# Class: Natural Computing
#####

# Import relevant libraries
import numpy as np
import random
import math

#####
# Simple Hill Climb (alg 3.1)
# initialize x
# loop:
# x' = delx + x
# if x' is better than x, x = x'
#####
def simple_climb( xtry ):
    xbest1 = 1.01 # xbest1, 2, 3 are used to calculate xavg
    xbest2 = 1.01
    xbest3 = 1.01
    xavg = 1.00 # calculate avg
    delx = 0.01 # perterb x

    ybest = 0.0 # initialize ybest for comparison

    i = 0 # loop counter

    # stop at 1000 iterations or no significant improvement is made
    while (i < 1000) and (xavg > 0.1) :

        # evaluate x
        ytry = 2.0 ** (-2.0 * ( ( xtry - 0.1 ) / 0.9 ) ** 2.0) * math.sin( 5.0 * math.pi

        # if it's better, remember that and make record new 'bests'
        if ytry > ybest:
            ybest = ytry
            xbest3 = xbest2
            xbest2 = xbest1
            xbest1 = xtry
            xavg = ( xbest1 + xbest2 + xbest3 ) / 3.0

            xtry = xtry + delx
            i += 1

    return xbest1

# initialize x
xtry = random.random()
```

```

# run hill climb
xbest1 = simple_climb( xtry )

print 'Simple Hill Climb', '\n', 'X = ', xbest1

#####
#               Iterated Hill Climb (alg 3.1)
# initialize best
# loop:
#   initialize x
#   eval(x) via simple hill climb
#   if x' is better than best, best = x
#####

best = 0.0 # initialize best
i = 0      # loop counter

besteval = 2.0 ** ( -2.0 * ( ( best - 0.1 ) / 0.9 ) ** 2.0 ) * math.sin( 5.0 * math.pi

while (i < 1000) :
    # initialize x
    x = random.random()

    # run hill climb on x
    x = simple_climb( x )

    # evaluate x
    xeval = 2.0 ** ( -2.0 * ( ( x - 0.1 ) / 0.9 ) ** 2.0 ) * math.sin( 5.0 * math.pi *

    # if xeval is better than best, store xeval as best
    if( xeval > besteval ):
        best = x
        besteval = xeval

    i += 1

print '\n', 'Iterated Hill Climb', '\n', 'X = ', best

#####
#               Stochastic Hill Climb (alg 3.3)
# initialize x
# eval x
# loop:
#   x' = random delx + x
#   eval x'
#   if random[0,1) < ( 1 / ( 1 + exp[ ( eval x - eval x' ) / T] ) ),
#       x = x'
#####

x = random.random() # initialize x
i = 0               # loop counter

# evaluate x

```

```
xeval = 2.0 ** ( -2.0 * ( ( x - 0.1 ) / 0.9 ) ** 2.0 ) * math.sin( 5.0 * math.pi * x )

while (i < 1000) :
    # randomly perturb x'
    xp = x + np.random.normal( 0, 0.2 )

    # check x', must stay between 0 and 1
    while not ( xp <= 1 and xp >= 0 ):
        xp = x + np.random.normal( 0, 0.2 )

    # evaluate x'
    xpeval = 2.0 ** ( -2.0 * ( ( xp - 0.1 ) / 0.9 ) ** 2.0 ) * math.sin( 5.0 * math.pi

    # calculate probability of accepting x'
    p = 1.0 / ( 1.0 + math.exp( (xeval - xpeval) / 0.01 ) )

    if random.random() < p :
        x = xp
        xeval = xpeval

    i += 1

print '\n', 'Stochastic Hill Climb', '\n', 'X = ', x
```

Listing B.5: BoneMarrow.py

```

import random
import copy
from collections import deque

class BoneMarrow:

    def __init__(self, repair_func):
        self.libraries = []
        self.mutators = []
        self.repair_func = repair_func

    def add_complete_gene_library(self, library):
        self.libraries.append(library)

    def add_gene_library(self, gene_generator, size):
        library = []
        for i in range(size):
            library.append(gene_generator())
        self.add_complete_gene_library(library)

    def create_chromosome(self):
        if len(self.libraries) != 0:
            chromosome = copy.deepcopy(self.libraries[0][random.randint(0, len(self.libraries[0]) - 1)])
            if len(self.libraries) == 1:
                return chromosome
            for i in range(1, len(self.libraries)):
                chromosome.extend(copy.deepcopy(self.libraries[i][random.randint(0, len(self.libraries[i]) - 1)]))
            return self.repair_func(chromosome)

    def create_chromosomes(self, count):
        chromosomes = []
        for i in range(count):
            chromosomes.append(self.create_chromosome())
        return chromosomes

if __name__ == '__main__':
    def random_generator():
        unused = range(1, 33)
        gene = []
        for i in range(4):
            index = random.randint(0, len(unused) - 1)
            gene.append(unused.pop(index))
        return gene

    def guided_generator():
        #Optimal path seen in Figure 6.24
        path = [1, 6, 9, 15, 18, 16, 10, 2, 7, 3, 11, 19, 23, 26, 30, 32, 29, 22, 25, 27, 24, 28, 31, 5, 8, 4, 12, 13, 14, 17, 20, 21]
        index = random.randint(0, len(path) - 1 - 4)
        gene = []
        for i in range(4):
            gene.append(path[index + i])
        return gene

```

```
def tsp_repair(chromosome):
    full_set = deque(range(1, 33))
    used = []
    for i in chromosome:
        if i in full_set:
            full_set.remove(i)
    for i in range(len(chromosome)):
        if chromosome[i] in used:
            chromosome[i] = full_set.popleft()
        used.append(chromosome[i])
    return chromosome

rand_bm = BoneMarrow(tsp_repair)
guided_bm = BoneMarrow(tsp_repair)

for i in range(8):
    rand_bm.add_gene_library(random_generator, 4)
    guided_bm.add_gene_library(guided_generator, 4)

print rand_bm.create_chromosomes(32)
print guided_bm.create_chromosomes(32)
```