# CSC 492/592 BioInspired Computing

## Homework 3

Mack Smith Derek Stotz

May 4, 2015

# Contents

# List of Figures

# List of Tables

# Document Preparation and Updates

Current Version [1.0.0]

*Prepared By:*
*Ian Carlson*
*Derek Stoz*

***Revision History***

| ***Date*** | ***Author*** | ***Version*** | ***Comments*** |
|---|---|---|---|
| ***2/2/15*** | *Ian Carlson and Derek Stoz* | *1.0.0* | *Wrote the Whole Thing* |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

1

# Immunocomputing - Text Chapter 6

## 1.1 Problem 6.1

Use a bone marrow algorithm to define genes for gene libraries to be used to generate the inital population of a genetic algorithm to solve the TSP presented in CH 3 and CH 5 (figure 6.24).

### 1.1.1 Problem Information

As described, this problem requested that we generate an initial population for a genetic algorithm to use when solving the travelling salesman problem depicted in figure 6.24 in the book. It also elaborated upon the description with the following (paraphrased):

Gene length Lg = 4, number of libraries n = 8, and library length (number of genes in each library) Ll = 4. As one gene from each library will be selected, the total chromosome length is L = Lg x n = 4 x 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.

A repair algorithm must be used in order to generate permutations of L integers while maintaining most of the genes intact. This is because the TSP problem has the contraint that no city can be visited more than once. to illustrate this problem and one possible way of solving it, consider the example presented in figure 6.25 in the book.

### 1.1.2 Algorithm Description

Bone marrow algorithms are simple techniques used primarily to build antibodies out of gene libraries for artificial immune systems. Generally, they can also be utilized to build genomes out of a collection of random genotypes stored in such a gene library.

Building a list of cities using the bone marrow algorithm was straightforwards. First of all, we began by creating a random set of gene libraries by randomly sampling stretches of cities (of length 4, of course) from a hardcoded optimal route, as specified in the problem. Next, we used these gene libraries to build enough routes to fill a population, taking a random stretch of cities from each gene library per route. After each route was built, however, we applied a repair algorithm in order to fit the constraints of the travelling salesman problem. This involved replacing all repeated cities with randomly ordered cities which were unused before the repairing. In the end, the python script prints out all routes in the initial population. If one were to then implement a GA for the travelling salesman problem, the population could be used to begin selection and breeding.

### 1.1.3 Results

As it was outside of the scope of the problem, we did not evaluate the routes generated by the bone marrow algorithm. However, the variety was apparent while the stretches of pre-built genes were clearly present in the genomes.

# 2

# Immunocomputing - Text Chapter 6

## 2.1 Problem 6.1

Use a bone marrow algorithm to define genes for gene libraries to be used to generate the inital population of a genetic algorithm to solve the TSP presented in CH 3 and CH 5 (figure 6.24).

### 2.1.1 Problem Information

As described, this problem requested that we generate an initial population for a genetic algorithm to use when solving the travelling salesman problem depicted in figure 6.24 in the book. It also elaborated upon the description with the following (paraphrased):

Gene length Lg = 4, number of libraries n = 8, and library length (number of genes in each library) Ll = 4. As one gene from each library will be selected, the total chromosome length is L = Lg x n = 4 x 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.

A repair algorithm must be used in order to generate permutations of L integers while maintaining most of the genes intact. This is because the TSP problem has the contraint that no city can be visited more than once. to illustrate this problem and one possible way of solving it, consider the example presented in figure 6.25 in the book.

### 2.1.2 Algorithm Description

Bone marrow algorithms are simple techniques used primarily to build antibodies out of gene libraries for artificial immune systems. Generally, they can also be utilized to build genomes out of a collection of random genotypes stored in such a gene library.

Building a list of cities using the bone marrow algorithm was straightforwards. First of all, we began by creating a random set of gene libraries by randomly sampling stretches of cities (of length 4, of course) from a hardcoded optimal route, as specified in the problem. Next, we used these gene libraries to build enough routes to fill a population, taking a random stretch of cities from each gene library per route. After each route was built, however, we applied a repair algorithm in order to fit the constraints of the travelling salesman problem. This involved replacing all repeated cities with randomly ordered cities which were unused before the repairing. In the end, the python script prints out all routes in the initial population. If one were to then implement a GA for the travelling salesman problem, the population could be used to begin selection and breeding.

### 2.1.3 Results

As it was outside of the scope of the problem, we did not evaluate the routes generated by the bone marrow algorithm. However, the variety was apparent while the stretches of pre-built genes were clearly present in the genomes.

# 3

# Immunocomputing - Text Chapter 6

## 3.1 Problem 6.1

Use a bone marrow algorithm to define genes for gene libraries to be used to generate the inital population of a genetic algorithm to solve the TSP presented in CH 3 and CH 5 (figure 6.24).

### 3.1.1 Problem Information

As described, this problem requested that we generate an initial population for a genetic algorithm to use when solving the travelling salesman problem depicted in figure 6.24 in the book. It also elaborated upon the description with the following (paraphrased):

Gene length Lg = 4, number of libraries n = 8, and library length (number of genes in each library) Ll = 4. As one gene from each library will be selected, the total chromosome length is L = Lg x n = 4 x 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.

A repair algorithm must be used in order to generate permutations of L integers while maintaining most of the genes intact. This is because the TSP problem has the contraint that no city can be visited more than once. to illustrate this problem and one possible way of solving it, consider the example presented in figure 6.25 in the book.

### 3.1.2 Algorithm Description

Bone marrow algorithms are simple techniques used primarily to build antibodies out of gene libraries for artificial immune systems. Generally, they can also be utilized to build genomes out of a collection of random genotypes stored in such a gene library.

Building a list of cities using the bone marrow algorithm was straightforwards. First of all, we began by creating a random set of gene libraries by randomly sampling stretches of cities (of length 4, of course) from a hardcoded optimal route, as specified in the problem. Next, we used these gene libraries to build enough routes to fill a population, taking a random stretch of cities from each gene library per route. After each route was built, however, we applied a repair algorithm in order to fit the constraints of the travelling salesman problem. This involved replacing all repeated cities with randomly ordered cities which were unused before the repairing. In the end, the python script prints out all routes in the initial population. If one were to then implement a GA for the travelling salesman problem, the population could be used to begin selection and breeding.

### 3.1.3 Results

As it was outside of the scope of the problem, we did not evaluate the routes generated by the bone marrow algorithm. However, the variety was apparent while the stretches of pre-built genes were clearly present in the genomes.

# 4

# Immunocomputing - Text Chapter 6

## 4.1 Problem 6.1

Use a bone marrow algorithm to define genes for gene libraries to be used to generate the inital population of a genetic algorithm to solve the TSP presented in CH 3 and CH 5 (figure 6.24).

### 4.1.1 Problem Information

As described, this problem requested that we generate an initial population for a genetic algorithm to use when solving the travelling salesman problem depicted in figure 6.24 in the book. It also elaborated upon the description with the following (paraphrased):

Gene length Lg = 4, number of libraries n = 8, and library length (number of genes in each library) Ll = 4. As one gene from each library will be selected, the total chromosome length is L = Lg x n = 4 x 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.

A repair algorithm must be used in order to generate permutations of L integers while maintaining most of the genes intact. This is because the TSP problem has the contraint that no city can be visited more than once. to illustrate this problem and one possible way of solving it, consider the example presented in figure 6.25 in the book.

### 4.1.2 Algorithm Description

Bone marrow algorithms are simple techniques used primarily to build antibodies out of gene libraries for artificial immune systems. Generally, they can also be utilized to build genomes out of a collection of random genotypes stored in such a gene library.

Building a list of cities using the bone marrow algorithm was straightforwards. First of all, we began by creating a random set of gene libraries by randomly sampling stretches of cities (of length 4, of course) from a hardcoded optimal route, as specified in the problem. Next, we used these gene libraries to build enough routes to fill a population, taking a random stretch of cities from each gene library per route. After each route was built, however, we applied a repair algorithm in order to fit the constraints of the travelling salesman problem. This involved replacing all repeated cities with randomly ordered cities which were unused before the repairing. In the end, the python script prints out all routes in the initial population. If one were to then implement a GA for the travelling salesman problem, the population could be used to begin selection and breeding.

### 4.1.3 Results

As it was outside of the scope of the problem, we did not evaluate the routes generated by the bone marrow algorithm. However, the variety was apparent while the stretches of pre-built genes were clearly present in the genomes.

5

# Immunocomputing - Text Chapter 6

## 5.1 Problem 6.1

Use a bone marrow algorithm to define genes for gene libraries to be used to generate the inital population of a genetic algorithm to solve the TSP presented in CH 3 and CH 5 (figure 6.24).

### 5.1.1 Problem Information

As described, this problem requested that we generate an initial population for a genetic algorithm to use when solving the travelling salesman problem depicted in figure 6.24 in the book. It also elaborated upon the description with the following (paraphrased):

Gene length Lg = 4, number of libraries n = 8, and library length (number of genes in each library) Ll = 4. As one gene from each library will be selected, the total chromosome length is L = Lg x n = 4 x 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.

A repair algorithm must be used in order to generate permutations of L integers while maintaining most of the genes intact. This is because the TSP problem has the contraint that no city can be visited more than once. to illustrate this problem and one possible way of solving it, consider the example presented in figure 6.25 in the book.

### 5.1.2 Algorithm Description

Bone marrow algorithms are simple techniques used primarily to build antibodies out of gene libraries for artificial immune systems. Generally, they can also be utilized to build genomes out of a collection of random genotypes stored in such a gene library.

Building a list of cities using the bone marrow algorithm was straightforwards. First of all, we began by creating a random set of gene libraries by randomly sampling stretches of cities (of length 4, of course) from a hardcoded optimal route, as specified in the problem. Next, we used these gene libraries to build enough routes to fill a population, taking a random stretch of cities from each gene library per route. After each route was built, however, we applied a repair algorithm in order to fit the constraints of the travelling salesman problem. This involved replacing all repeated cities with randomly ordered cities which were unused before the repairing. In the end, the python script prints out all routes in the initial population. If one were to then implement a GA for the travelling salesman problem, the population could be used to begin selection and breeding.

### 5.1.3 Results

As it was outside of the scope of the problem, we did not evaluate the routes generated by the bone marrow algorithm. However, the variety was apparent while the stretches of pre-built genes were clearly present in the genomes.

# A

# Code

```
#!/user/bin/python

"""
H2C3P1 - Deterministic Hill Climbing

Authors: Derek Stotz, Ian Carlson
Date: 3/16/2015
Course: Natural Computing
Prof: Dr. Jeff McGough
Usage:
  python3 deterministic_hillclimb.py <max iterations> <resolution> <start x>
"""

from math import *
from random import *
from sys import *

DISTRIBUTION = "UNIFORM"
DEBUG = False

# prints a single point in the search space and its fitness
# x - the point in the search space
def hc_print(i, x, f):
  print("i: " + str(i) + "\tx:" + str(x) + "\tf(x):" + str(f))

# Looks left and right from x by amount res
# x - the current location
# f - the current fitness
# res - step size
# returns the new location, new fitness, and true
# or the old location, fitness and false
def steepest_ascent(x,f,res):
  lx = x-res
  rx = x+res
  lf = fit(lx)
  rf = fit(rx)
  newx = x
  newf = f
  improvement = False
```

```
    if lf > newf:
        newx = lx
        newf = lf
        improvement = True
    if rf > newf:
        newx = rx
        newf = rf
        improvement = True
    return [newx,newf,improvement]


# returns the fitness value of x, a point in the search space
# x - the point in the search space
def fit(x):
    return (2 ** ( -2 * ( (x-.1)/.9 ) ** 2 ) * (sin(5*pi*x)) ** 6)

# the main function, setting up environment and evaluating results
# max_i - the maximum iteration
# res - he step size for searching
# start_x - the starting point
def hillclimb(max_i, res, start_x):
    x = start_x
    f = fit(x)

    for i in range(0, max_i):
        if DEBUG:
            hc_print(i, x, f)
        [x,f,improvement] = steepest_ascent(x,f,res)
        if not improvement:
            return [x,f,i]
    return [x, f, i]


if __name__ == "__main__":

    seed()
    max_i = 10000
    res = 0.00001
    #Initialize a random start [-1,1)
    start_x = (random()-0.5)*2;

    if(len(argv)>3):
        start_x = float(argv[3])
    if(len(argv)>2):
        res = float(argv[2])
    if(len(argv)>1):
        max_i = int(argv[1])

    results = hillclimb(max_i, res, start_x)
    best_x = results[0]
    best_f = results[1]
    iterations = results[2] + 1
```

```
print("The best x found after " + str(iterations) + " iterations was " + str(best_x) + "
    with a value of " + str(best_f) + ".")
```