1. import math;
2. import random;
3. from bitarray import bitarray;
4. import sys;
6. mutate\_prob = 0.05 # Small probability with which mutation will occur.
8. class Population(object):
9. population\_list = [] # List of individuals: binary strings which represent tentative solutions.
10. fitness\_list = [] # List of the fitness values of each individual in the population.
11. pop\_size = 0 # Population Size.
12. board\_size = 0 # Size of problem. 'N' in N-Queen.
13. pos\_bits\_size = 0 # Number of bits required to represent each vertical position.
14. indv\_size = 0 # The total size of an 'induvidual' binary string.
15. def \_\_init\_\_(self, pop\_size, board\_size):
16. self.pop\_size = pop\_size
17. self.board\_size = board\_size
18. self.population\_list = []
19. self.pos\_bits\_size = int(math.ceil(math.log(board\_size-1))) + 1
20. self.indv\_size = board\_size \* self.pos\_bits\_size
22. def genPopulation(self):
23. """Generates a population of bitarray strings (individuals) to solve the N-Queen problem, here N = 'board\_size'.
24. 'pop\_size' parameter decides the number of bitarray strings (individuals) generated by this function.
25. Creates a list of bitarrray strings (individuals) representing the population for solving the N-Queen problem.
26. 'population\_list' parameter holds the list of generated individuals."""
27. self.population\_list = []
28. for i in xrange(0, self.pop\_size):
29. individual = bitarray(self.indv\_size)
30. # Loop for randomizing the 'individual' string.
31. for j in xrange(0, self.board\_size):
32. vert\_pos = random.randint(0, self.board\_size-1)
33. vert\_pos\_bitnum = toBitArray(vert\_pos, self.pos\_bits\_size)
34. # print "\t\t", j, vert\_pos\_bitnum, vert\_pos
35. for k in range(0, self.pos\_bits\_size):
36. individual[j \* self.pos\_bits\_size + k] = vert\_pos\_bitnum[k]
37. self.population\_list.append(individual)
38. # print "\t", i, individual
40. def computeFitnessList(self,fitnessFunction):
41. """Populates the fitness list with fitness function values of co-responding entries in the population list."""
42. self.fitness\_list = []
43. cmlsum = 0
44. for individual in self.population\_list:
45. cmlsum = cmlsum + fitnessFunction(individual, self.board\_size, self.pos\_bits\_size)
46. self.fitness\_list.append(cmlsum)
48. def findInFitnessList(self, key):
49. """Binary Search for finding appropriate index for key in fitness list."""
50. low = 0
51. high = len(self.fitness\_list)
52. mid = 0
53. while(low <= high):
54. mid = (low + high)/2
55. if key > self.fitness\_list[mid]:
56. low = mid + 1
57. elif key < self.fitness\_list[mid]:
58. high = mid - 1
59. else:
60. break
61. if low > high:
62. return low
63. else:
64. return mid

67. def toBitArray(number, size):
68. """Converts a number in int format to a bitarray of length 'size', in binary representation.
69. Returns the bitarray."""
70. temp\_bitnum = bitarray(64)
71. count = 0
72. number = number & 0xFFFFFFFFFFFFFFFF # enforces the number to be 64 bit.
73. while count < size:
74. temp\_bitnum[63 - count] = (number % 2)
75. # print "digit ", count, " : ", (number % 2)
76. number = number >> 1
77. count = count + 1
78. return temp\_bitnum[-size:]
80. def fromBitArray(bitnum):
81. """Converts a bitarray in binary format to a number in int format, to its co-responding decimal representation.
82. Returns the number."""
83. number = 0 & 0xFFFFFFFFFFFFFFFF # enforces the number to be 64 bit.
84. idx = len(bitnum) - 1
85. number = number + bitnum[idx]
86. idx = idx - 1
87. count = 1
88. while idx != -1:
89. number = number + (bitnum[idx] << count)
90. count = count + 1
91. idx = idx - 1
92. return number
94. def fitnessFunction(individual, board\_size, pos\_bits\_size):
95. """Calculates the finess value of an individual and returns it.
96. Has a computational complexity of O(1) per piece."""
97. right\_diag = [0] \* (2 \* board\_size - 1)
98. left\_diag = [0] \* (2 \* board\_size - 1)
99. vertical = [0] \* board\_size
100. conflicts = 0
101. idx = 0
102. while idx < board\_size:
103. # print "idx: ",idx,individual[idx \* pos\_bits\_size : idx \* pos\_bits\_size + pos\_bits\_size]
104. vpos = fromBitArray(individual[idx \* pos\_bits\_size : idx \* pos\_bits\_size + pos\_bits\_size])
105. # print "vpos: ", vpos + 1
106. if vertical[vpos] != 0:
107. conflicts = conflicts + vertical[vpos]
108. vertical[vpos] = vertical[vpos] + 1
109. if left\_diag[vpos + idx] != 0:
110. conflicts = conflicts + left\_diag[vpos + idx]
111. left\_diag[vpos + idx] = left\_diag[vpos + idx] + 1
112. if right\_diag[vpos + board\_size - idx - 1] != 0:
113. conflicts = conflicts + right\_diag[vpos + board\_size - idx - 1]
114. right\_diag[vpos + board\_size - idx - 1] = right\_diag[vpos + board\_size - idx - 1] + 1
115. idx = idx + 1
116. return (board\_size \* (board\_size - 1))/2 - conflicts
118. def geneticAlgorithm(population, fitnessFunction):
119. global mutate\_prob
120. child = None
121. condition = True
122. while condition:
123. population.computeFitnessList(fitnessFunction)
124. new\_pop = []
125. for i in xrange(0,len(population.population\_list)):
126. parent\_x = None
127. parent\_y = None
128. (parent\_x, parent\_y) = randomSelection(population)
129. child = reproduce(parent\_x, parent\_y, population)
130. if mutate\_prob > random.random():
131. mutate(child, population)
132. new\_pop.append(child)
133. population.population\_list = new\_pop
134. condition = (fitnessFunction(child, population.board\_size, population.pos\_bits\_size) != \
135. (population.board\_size \* (population.board\_size - 1))/2)
136. # check condition
137. return child
139. def randomSelection(population):
140. rand\_sel\_x = random.randint(1, population.fitness\_list[len(population.fitness\_list) - 1])
141. parent\_x\_idx = population.findInFitnessList(rand\_sel\_x)
142. range\_rem = population.fitness\_list[parent\_x\_idx]
143. if rand\_sel\_x > population.fitness\_list[0]:
144. range\_rem = range\_rem - population.fitness\_list[parent\_x\_idx - 1]
145. rand\_sel\_y = random.randint(1, population.fitness\_list[len(population.fitness\_list) - 1] - range\_rem)
146. if rand\_sel\_y >= rand\_sel\_x:
147. rand\_sel\_y = rand\_sel\_y + range\_rem
148. parent\_y\_idx = population.findInFitnessList(rand\_sel\_y)
149. parent\_x = population.population\_list[parent\_x\_idx]
150. parent\_y = population.population\_list[parent\_y\_idx]
151. return (parent\_x, parent\_y)
153. def reproduce(parent\_x, parent\_y, population):
154. crossover\_pt = random.randint(1, population.board\_size - 1)
155. # print crossover\_pt
156. return parent\_x[:crossover\_pt \* population.pos\_bits\_size] + \
157. parent\_y[crossover\_pt \* population.pos\_bits\_size:]
159. def mutate(child, population):
160. rand\_idx = random.randint(0, population.board\_size - 1)
161. rand\_vpos = random.randint(0, population.board\_size - 1)
162. temp\_bitnum = toBitArray(rand\_vpos, population.pos\_bits\_size)
163. # print "mutate: ", rand\_idx, temp\_bitnum
164. for i in range(0, population.pos\_bits\_size):
165. child[rand\_idx \* population.pos\_bits\_size + i] = temp\_bitnum[i]
167. new\_pop = Population(int(sys.argv[1]),int(sys.argv[2]))
168. new\_pop.genPopulation()
169. result = geneticAlgorithm(new\_pop,fitnessFunction)
170. for i in range(0,new\_pop.board\_size):
171. print fromBitArray(result[i \* new\_pop.pos\_bits\_size : i \* new\_pop.pos\_bits\_size + new\_pop.pos\_bits\_size]) + 1,