

Branch: master ▾

[Python_Scripts](#) / [Python_Scripts](#) / [Coursework_2](#) / [rw.py](#)

Find file

Copy pa

 Candidate Number: 091388 Ammended DocStrings for rw.py

bac5593 2 minutes ag

1 contributor

310 lines (216 sloc) 8.77 KB

```
1 from Coursework_2.hvg import *
2 from Coursework_2.choose_vertex import choose_vertex
3
4
5 def get_graph_from_series(length=25):
6     """
7     Generates adjacency matrix from a numerical series of size 'length'
8     from logistic map with 0.1 as initial condition and parameter set to 4
9
10    :param length: Size of the series to be generated
11    :return: Adjacency matrix produced from series
12    """
13
14    series = logistic_map(0.1, length, 4)
15    matrix = horizontal_visibility_graph(series)
16
17    return matrix
18
19
20 def random_walk(adj_matrix, steps=1000, biased=False, alpha=1.0):
21     """
22     Performs either biased or non biased random walks on graphs encoded in an
23     adjacency matrix, returning a list of visited vertices
24
25    :param adj_matrix: Horizontal visibility graph encoded in matrix format
26    :param steps: Number of y coordinates to be produced
27    :param biased: Decides whether walk is biased or not
28    :param alpha: Parameter used in random walk calculations
29    :return: A list of visited vertices on random walk
30    """
31
32    visited_vertices = []
33
34    if biased:
35
36        # CREATE BIASED TRANSITION MATRIX
37        transition_matrix = create_transition_matrix(adj_matrix, True, alpha)
38
39        # CREATE BIASED INITIAL VERTEX DISTRIBUTION
40        initial_vertex_distribution = find_initial_vertex_dist(adj_matrix, True, alpha)
41
42        # ADD FIRST VERTEX TO LIST
43        visited_vertices.append(choose_vertex(initial_vertex_distribution))
44
45        # ADD REST OF VERTICES TO LIST
46        for i in range(1, steps):
47            prev = visited_vertices[len(visited_vertices) - 1]
48            visited_vertices.append(choose_vertex(transition_matrix[prev]))
49
50        return visited_vertices
51
52    else:
53
54        # CREATE TRANSITION MATRIX
55        transition_matrix = create_transition_matrix(adj_matrix)
56
57        # CREATE INITIAL VERTEX DISTRIBUTION
58        initial_vertex_distribution = find_initial_vertex_dist(adj_matrix)
```

```

59
60     # ADD FIRST VERTEX TO LIST
61     visited_vertices.append(choose_vertex(initial_vertex_distribution))
62
63     # ADD REST OF VERTICES TO LIST
64     for i in range(1, steps):
65         prev = visited_vertices[len(visited_vertices) - 1]
66         visited_vertices.append(choose_vertex(transition_matrix[prev]))
67
68     return visited_vertices
69
70
71 def create_transition_matrix(adj_matrix, biased=False, alpha=0):
72     """
73     Creates a biased/unbiased transition matrix from a given adjacency matrix
74
75     :param adj_matrix: A horizontal visibility graph encoded in an
76         adjacency matrix
77     :param biased: Decides whether to produce a biased/unbiased
78         transition matrix
79     :param alpha: Parameter used in random walk calculations
80     :return: A transition matrix as a list of lists
81     """
82
83     # BUILD MATRIX FILLED WITH 0'S
84     t_matrix = [[0 for j in range(len(adj_matrix))] for i in
85                 range(len(adj_matrix))]
86
87     if biased:
88
89         for i in range(len(adj_matrix)):
90             for j in range(len(adj_matrix)):
91
92                 # CALCULATE NUMERATOR
93                 a_ij = adj_matrix[i][j]
94                 d_alpha = count_edges(adj_matrix, j) ** alpha
95                 numerator = a_ij * d_alpha
96
97                 # CALCULATE DENOMINATOR
98                 denominator = 0
99                 for k in range(len(adj_matrix)):
100                     a_ik = adj_matrix[i][k]
101                     d_j = count_edges(adj_matrix, k)
102                     denominator = denominator + (a_ik * d_j ** alpha)
103
104                 # OVERRIDE VALUE
105                 if denominator == 0:
106                     continue
107                 else:
108                     t_matrix[i][j] = numerator / denominator
109
110     else:
111
112         for i in range(len(adj_matrix)):
113
114             probability = 1/count_edges(adj_matrix, i)
115
116             # OVERRIDE VALUES
117             for j in range(len(adj_matrix)):
118                 if adj_matrix[i][j] == 1:
119                     t_matrix[i][j] = probability
120
121     return t_matrix
122
123
124 def count_edges(adj_matrix, row_index):
125     """
126     Counts the edges in a row of an adjacency matrix
127
128     :param adj_matrix: Horizontal visibility graph encoded in matrix format
129     :param row_index: The index of the row for edges to be counted

```

```

130 :return: The number of edges present the given row
131 """
132 row = adj_matrix[row_index]
133 count = 0
134
135 for i in row:
136     if i != 0:
137         count += 1
138
139 return count
140
141
142 def find_initial_vertex_dist(adj_matrix, biased=False, alpha=0):
143     """
144     Produces a list of initial vertex distributions for
145         biased and unbiased random walks
146
147     :param adj_matrix: Horizontal visibility graph encoded in matrix form
148     :param biased: Decides whether to calculate for biased / unbiased walk
149     :param alpha: Parameter used in random walk calculations
150     :return: A list containing the probability distribution for the initial vertex
151     """
152     total_ones = 0
153     ones_in_row = []
154     initial_vertices = []
155     numerators = []
156
157     if biased:
158
159         for i in range(len(adj_matrix)):
160
161             # C CALCULATION
162             d_j_alpha = count_edges(adj_matrix, i) ** alpha
163
164             # NUMERATOR CALCULATIONS
165             c_i = count_edges(adj_matrix, i) * d_j_alpha
166             d_i_alpha = count_edges(adj_matrix, i) ** alpha
167             numerator = c_i * d_i_alpha
168             numerators.append(numerator)
169
170             # DENOMINATOR CALCULATION
171             denominator = sum(numerators)
172
173             # POPULATE INITIAL VERTICES
174             for i in range(len(adj_matrix)):
175                 initial_vertices.append(numerators[i] / denominator)
176
177         return initial_vertices
178
179     else:
180
181         # CALCULATE PROBABILITY
182         for i in range(len(adj_matrix)):
183             total_ones = total_ones + count_edges(adj_matrix, i)
184             ones_in_row.append(count_edges(adj_matrix, i))
185
186         # POPULATE INITIAL VERTICES
187         for i in range(len(adj_matrix)):
188             initial_vertices.append(ones_in_row[i]/total_ones)
189
190     return initial_vertices
191
192
193 def print_triplets(visited_vertices, k=20):
194     """
195     Prints 'k' amount of triplets from a list of visited vertices
196
197     :param visited_vertices: A list of vertices visited on a random walk
198     :param k: The number of triplets to be produced
199     """
200

```

```

201     number_triplets_poss = len(visited_vertices) - 2
202
203     if number_triplets_poss >= k:
204
205         # PRINT TRIPLETS
206         for i in range(k):
207             print(visited_vertices[i:i+3])
208
209     else:
210
211         # PRINT MOST POSSIBLE TRIPLETS WITH ERROR MESSAGE
212         for i in range(number_triplets_poss):
213             print(visited_vertices[i:i+3])
214             print("\nOnly", number_triplets_poss, "triplets were possible")
215
216
217 def verify_equality(adj_matrix):
218     """
219     Checks that the transition matrix for an unbiased walk is equivalent to
220     the transition matrix of a biased walk with alpha = 0
221
222     :param adj_matrix: Horizontal visibility graph encoded in matrix form
223     """
224
225     if create_transition_matrix(adj_matrix) == create_transition_matrix(adj_matrix, True, 0):
226         print("True")
227     else:
228         print("False")
229
230
231 def histogram(adj_matrix):
232     """
233     Prints histogram of vertex degrees of a generated graph
234
235     :param adj_matrix: Horizontal visibility graph encoded in matrix form
236     """
237
238     for i in range(len(adj_matrix)):
239         print("Vertex ID", i, ":", "*" * count_edges(adj_matrix, i))
240
241
242 def count_vertex_occurrence(visited_vertices):
243     """
244     Creates a dictionary containing all the vertices visited and the frequency
245     of which they were visited on a random walk
246
247     :param visited_vertices: A list of vertices visited on a random walk
248     :return: A dictionary containing vertices and frequency of which
249             they were visited
250     """
251
252     dictionary = {}
253     items_present = []
254
255     # FIND ITEMS PRESENT
256     for i in visited_vertices:
257         if i not in items_present:
258             items_present.append(i)
259
260     items_present.sort()
261
262     # ADD TO DICTIONARY
263     for i in items_present:
264         dictionary[i] = visited_vertices.count(i)
265
266     # FORMAT PRINT
267     for vertex, times_visited in dictionary.items():
268         print("Vertex: ", vertex, ", Times Visited: ", times_visited)
269
270
271 # MAIN PROGRAM

```

```
272 if __name__ == "__main__":
273
274     # CREATE SERIES
275     graph_from_series = get_graph_from_series()
276
277     # CREATE WALKS
278     non_biased = random_walk(graph_from_series, 200, False, 5.0)
279     biased = random_walk(graph_from_series, 200, True, 5.0)
280
281     print("NON BIASED RANDOM WALK")
282     print("Visited Vertices: ", non_biased, end="\n\n")
283
284     print("BIASED RANDOM WALK")
285     print("Visited Vertices: ", biased, end="\n\n")
286
287     print("NON BIASED TRIPLETS")
288     print_triplets(non_biased)
289     print("")
290
291     print("BIASED TRIPLETS")
292     print_triplets(biased)
293     print("")
294
295     print("HISTOGRAM")
296     histogram(graph_from_series)
297     print("")
298
299     print("NON BIASED DICTIONARY")
300     count_vertex_occurrence(non_biased)
301     print("")
302
303     print("BIASED DICTIONARY")
304     count_vertex_occurrence(biased)
305     print("")
306
307
308
309
```