310 lines (216 sloc) 8.77 KB

```python
from Coursework_2.hvg import *
from Coursework_2.choose_vertex import choose_vertex


def get_graph_from_series(length=25):
    """
    Generates adjacency matrix from a numerical series of size 'length'
        from logistic map with 0.1 as initial condition and parameter set to 4

    :param length: Size of the series to be generated
    :return: Adjacency matrix produced from series
    """

    series = logistic_map(0.1, length, 4)
    matrix = horizontal_visibility_graph(series)

    return matrix


def random_walk(adj_matrix, steps=1000, biased=False, alpha=1.0):
    """
    Performs either biased or non biased random walks on graphs encoded in an
        adjacency matrix, returning a list of visited vertices

    :param adj_matrix: Horizontal visibility graph encoded in matrix format
    :param steps: Number of y coordinates to be produced
    :param biased: Decides whether walk is biased or not
    :param alpha: Parameter used in random walk calculations
    :return: A list of visited vertices on random walk
    """

    visited_vertices = []

    if biased:

        # CREATE BIASED TRANSITION MATRIX
        transition_matrix = create_transition_matrix(adj_matrix, True, alpha)

        # CREATE BIASED INITIAL VERTEX DISTRIBUTION
        initial_vertex_distribution = find_initial_vertex_dist(adj_matrix, True, alpha)

        # ADD FIRST VERTEX TO LIST
        visited_vertices.append(choose_vertex(initial_vertex_distribution))

        # ADD REST OF VERTICES TO LIST
        for i in range(1, steps):
            prev = visited_vertices[len(visited_vertices) - 1]
            visited_vertices.append(choose_vertex(transition_matrix[prev]))

        return visited_vertices

    else:

        # CREATE TRANSITION MATRIX
        transition_matrix = create_transition_matrix(adj_matrix)

        # CREATE INITIAL VERTEX DISTRIBUTION
        initial_vertex_distribution = find_initial_vertex_dist(adj_matrix)
```

```python

        # ADD FIRST VERTEX TO LIST
        visited_vertices.append(choose_vertex(initial_vertex_distribution))

        # ADD REST OF VERTICES TO LIST
        for i in range(1, steps):
            prev = visited_vertices[len(visited_vertices) - 1]
            visited_vertices.append(choose_vertex(transition_matrix[prev]))

    return visited_vertices


def create_transition_matrix(adj_matrix, biased=False, alpha=0):
    """
    Creates a biased/unbiased transition matrix from a given adjacency matrix

    :param adj_matrix: A horizontal visibility graph encoded in an
        adjacency matrix
    :param biased: Decides whether to produce a biased/unbiased
        transition matrix
    :param alpha: Parameter used in random walk calculations
    :return: A transition matrix as a list of lists
    """

    # BUILD MATRIX FILLED WITH 0'S
    t_matrix = [[0 for j in range(len(adj_matrix))] for i in
                 range(len(adj_matrix))]

    if biased:

        for i in range(len(adj_matrix)):
            for j in range(len(adj_matrix)):

                # CALCULATE NUMERATOR
                a_ij = adj_matrix[i][j]
                d_alpha = count_edges(adj_matrix, j) ** alpha
                numerator = a_ij * d_alpha

                # CALCULATE DENOMINATOR
                denominator = 0
                for k in range(len(adj_matrix)):
                    a_ik = adj_matrix[i][k]
                    d_j = count_edges(adj_matrix, k)
                    denominator = denominator + (a_ik * d_j ** alpha)

                    # OVERRIDE VALUE
                    if denominator == 0:
                        continue
                    else:
                        t_matrix[i][j] = numerator / denominator

    else:

        for i in range(len(adj_matrix)):

            probability = 1/count_edges(adj_matrix, i)

            # OVERRIDE VALUES
            for j in range(len(adj_matrix)):
                if adj_matrix[i][j] == 1:
                    t_matrix[i][j] = probability

    return t_matrix


def count_edges(adj_matrix, row_index):
    """
    Counts the edges in a row of an adjacency matrix

    :param adj_matrix: Horizontal visibility graph encoded in matrix format
    :param row_index: The index of the row for edges to be counted
```

```python
    :return: The number of edges present the given row
    """
    row = adj_matrix[row_index]
    count = 0

    for i in row:
        if i != 0:
            count += 1

    return count


def find_initial_vertex_dist(adj_matrix, biased=False, alpha=0):
    """
    Produces a list of initial vertex distributions for
        biased and unbiased random walks

    :param adj_matrix: Horizontal visibility graph encoded in matrix form
    :param biased: Decides whether to calculate for biased / unbiased walk
    :param alpha: Parameter used in random walk calculations
    :return: A list containing the probability distribution for the initial vertex
    """
    total_ones = 0
    ones_in_row = []
    initial_vertices = []
    numerators = []

    if biased:

        for i in range(len(adj_matrix)):

            # C CALCULATION
            d_j_alpha = count_edges(adj_matrix, i) ** alpha

            # NUMERATOR CALCULATIONS
            c_i = count_edges(adj_matrix, i) * d_j_alpha
            d_i_alpha = count_edges(adj_matrix, i) ** alpha
            numerator = c_i * d_i_alpha
            numerators.append(numerator)

            # DENOMINATOR CALCULATION
            denominator = sum(numerators)

        # POPULATE INITIAL VERTICES
        for i in range(len(adj_matrix)):
            initial_vertices.append(numerators[i] / denominator)

        return initial_vertices

    else:

        # CALCULATE PROBABILITY
        for i in range(len(adj_matrix)):
            total_ones = total_ones + count_edges(adj_matrix, i)
            ones_in_row.append(count_edges(adj_matrix, i))

        # POPULATE INITIAL VERTICES
        for i in range(len(adj_matrix)):
            initial_vertices.append(ones_in_row[i]/total_ones)

    return initial_vertices


def print_triplets(visited_vertices, k=20):
    """
    Prints 'k' amount of triplets from a list of visited vertices

    :param visited_vertices: A list of vertices visited on a random walk
    :param k: The number of triplets to be produced
    """

```

```python
    number_triplets_poss = len(visited_vertices) - 2

    if number_triplets_poss >= k:

        # PRINT TRIPLETS
        for i in range(k):
            print(visited_vertices[i:i+3])

    else:

        # PRINT MOST POSSIBLE TRIPLETS WITH ERROR MESSAGE
        for i in range(number_triplets_poss):
            print(visited_vertices[i:i+3])
        print("\nOnly", number_triplets_poss, "triplets were possible")


def verify_equality(adj_matrix):
    """
    Checks that the transition matrix for an unbiased walk is equivalent to
        the transition matrix of a biased walk with alpha = 0

    :param adj_matrix: Horizontal visibility graph encoded in matrix form
    """

    if create_transition_matrix(adj_matrix) == create_transition_matrix(adj_matrix, True, 0):
        print("True")
    else:
        print("False")


def histogram(adj_matrix):
    """
    Prints histogram of vertex degrees of a generated graph

    :param adj_matrix: Horizontal visibility graph encoded in matrix form
    """

    for i in range(len(adj_matrix)):
        print("Vertex ID", i, ":", "*" * count_edges(adj_matrix, i))


def count_vertex_occurrence(visited_vertices):
    """
    Creates a dictionary containing all the vertices visited and the frequency
        of which they were visited on a random walk

    :param visited_vertices: A list of vertices visited on a random walk
    :return: A dictionary containing vertices and frequency of which
        they were visited
    """

    dictionary = {}
    items_present = []

    # FIND ITEMS PRESENT
    for i in visited_vertices:
        if i not in items_present:
            items_present.append(i)

    items_present.sort()

    # ADD TO DICTIONARY
    for i in items_present:
        dictionary[i] = visited_vertices.count(i)

    # FORMAT PRINT
    for vertex, times_visited in dictionary.items():
        print("Vertex: ", vertex, ", Times Visited: ", times_visited)


# MAIN PROGRAM
```

```python
if __name__ == "__main__":

    # CREATE SERIES
    graph_from_series = get_graph_from_series()

    # CREATE WALKS
    non_biased = random_walk(graph_from_series, 200, False, 5.0)
    biased = random_walk(graph_from_series, 200, True, 5.0)

    print("NON BIASED RANDOM WALK")
    print("Visited Vertices: ", non_biased, end="\n\n")

    print("BIASED RANDOM WALK")
    print("Visited Vertices: ", biased, end="\n\n")

    print("NON BIASED TRIPLETS")
    print_triplets(non_biased)
    print("")

    print("BIASED TRIPLETS")
    print_triplets(biased)
    print("")

    print("HISTOGRAM")
    histogram(graph_from_series)
    print("")

    print("NON BIASED DICTIONARY")
    count_vertex_occurrence(non_biased)
    print("")

    print("BIASED DICTIONARY")
    count_vertex_occurrence(biased)
    print("")
```