Lin_Yu HW3 HGEN 48800

April 14, 2024

1 Question 1

Idea: We can consider n professional westlers as n vertices and r pairs of rivalries as edges among the n vertices. Thus, we want all vertices that are related to each other to be different colors(red, blue). And we can use BFS Algorithm to implement this. BFS algorithm has running time O(n+r).

Pseudocode

Function designation (n: Integer, rivalries: List of Pairs)

```
Initialize graph as an empty adjacency list for n nodes
Initialize color as a list of size n with all values set to None (representing uncolored nodes)
Initialize queue as an empty list



//Construct the graph
For each pair (u, v) in rivalries
    Add v to the adjacency list of u
    Add u to the adjacency list of v

// Color the graph using BFS
For each node start from 0 to n-1
    If color[start] is None // Node has not been visited
        Append start to queue // Enqueue starting node
        Set color[start] to 0  // Color it Red (0)

        While queue is not empty
            Set current to the front of queue and remove it from the queue // Dequeue
            Set current_color to color[current]
            Set next_color to 1 - current_color // Determine alternate color

            For each neighbor in adjacency list of current
                If color[neighbor] is None // Neighbor is uncolored
                    Set color[neighbor] to next_color // Color the neighbor
                    Append neighbor to queue // Enqueue
                Else If color[neighbor] is equal to current_color // Conflict in coloring
                    Return False, empty list // Bipartite division is not possible

// If the loop completes without conflicts, prepare team assignments
```

```
Initialize blue_team_assignments as a list
Initialize red_team_assignments as a list
For each color_value in color
    If color_value is 1
        Append "Blue" to blue_team_assignments
    Else
        Append "Red" to red_team_assignments

Return True, team_assignments // Return successful bipartite division and assignments
```

End Function

2 Question 2

To count the number of paths from start vertex s to destination, based on what we have learnt in class, DFS algorthm could help us and to avoid counting the same path twice, we add

[1]:
```python
def count_paths(graph, start, end, memo):
    # Check if the result for this start is already computed
    if start in memo:
        return memo[start]

    # Base case: if start is the end, there's exactly one path to itself
    if start == end:
        return 1

    # Initialize the path count to 0
    path_count = 0

    # Visit all neighbors (since it's a DAG, no need to check for cycles)
    for neighbor in graph[start]:
        path_count += count_paths(graph, neighbor, end, memo)

    # Store the computed number of paths from start to end in the memoization␣
 ↪dictionary
    memo[start] = path_count
    return path_count

def count_all_paths(graph, s, d):
    # Create a dictionary to store the number of paths from each node to s
    memo = {}
    # Start the DFS from node u to s
    return count_paths(graph, s, d, memo)

# Example usage:
# Define a graph as an adjacency list
graph = {
    0: [1, 2 ,5],
    1: [3, 4],
    2: [3],
    3: [4, 5],
    4: [],
    5: []
}

# Count paths from vertex 0 to vertex 5
u = 1
s = 5
print("Number of paths from", u, "to", s, ":", count_all_paths(graph, u, s))
```

```
Number of paths from 1 to 5 : 1
```

3 Question 3

[2]:
```
# Part a

import random
random.seed(123)

def generate_genomo_sequence(length):
    # Define the possible characters in the DNA sequence
    bases = ['A', 'G', 'T', 'C']
    # Generate a random sequence of the specified length
    return ''.join(random.choice(bases) for _ in range(length))

# Test
test_sequence = generate_genomo_sequence(10)
print("Random DNA sequence:(test)", test_sequence)

# Simulate a genome of length 1000
sample_geno=generate_genomo_sequence(1000)
```

```
Random DNA sequence:(test) ATACTAACTT
```

[3]:
```
# Part b

# Simulate read data
def generate_reads(r_length, n_reads, geno):
    # Chopping a geno into small reads of length
    num = len(geno)
    reads = []
    if num < r_length:
        print("Genome length is too short for the specified read length.")
        return []
    else:
        for _ in range(n_reads):
        # Ensure the random start index allows for a full read of r_length
            i = random.randint(0, num - r_length)
            # slicing to get a substring from geno
            new_read = geno[i:i+r_length]
            reads.append(new_read)
    return reads

# Example genome sequence and function call

reads = generate_reads(25, 400, sample_geno)
#print("Generated reads:", reads)
```

[4]:
```
#Part c
from collections import defaultdict

def generate_kmers(read,k):
    #Return kmers for each read
    return [read[i:i+k] for i in range(len(read) - k + 1)]

# Construct the De Bruijn graph with k=10
def De_bruijn_graph(k,reads,n_reads):
    edges = defaultdict(set)
    nodes = set()

    for i in range(0,n_reads):
        read=reads[i] #Access i+1th read

        # Break all reads into k-mers
        kmers = generate_kmers(read, k)

        nodes.update(kmers) # add kmers generated from i+1th read to our nodes␣
 ↪set

        for j in range(len(kmers) - 1):
            edges[kmers[j]].add(kmers[j+1]) # add edge to our graph

    return dict(edges), nodes

graph, nodes = De_bruijn_graph(10,reads,400)
print("Number of nodes:", len(nodes))
```

```
Number of nodes: 986
```

[5]:
```
print("Some Edges in the De Bruijn Graph:", list(graph.items())[:1])
```

```
Some Edges in the De Bruijn Graph: [('TAGAACCGCA', {'AGAACCGCAC'})]
```

Part D Reference: https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/

[6]:
```
def find_eulerian_path(adj, nodes):
    """
    This function takes an adjacency list of a directed graph and prints the␣
 ↪Eulerian path
    or circuit using Hierholzer's algorithm, if it exists.
    """
    nodes = list(nodes)

    # Calculate in-degree and out-degree for each vertex
    in_degree = {i: 0 for i in nodes}
    out_degree = {i: 0 for i in nodes}
    edge_matrix = [[None] * len(nodes) for _ in range(len(nodes))]

    for node in nodes:
        for neighbor in adj[node]:
            out_degree[node] += 1
            in_degree[neighbor] += 1
            i = nodes.index(node) # Find the index of the node
            j = nodes.index(neighbor) # Find the index of the neighbor node
            edge_matrix[i][j] = 1    # If there is an edge, update the␣
 ↪corresponding cell to 1

    # List to store the path
    path = []
    stack = []

    # Start from a vertex with non-zero out-degree
    for vertex in nodes:
        if out_degree[vertex] != 0 and in_degree[vertex] == 0:
            start_vertex = vertex
            break

    stack.append(start_vertex)

    # Hierholzer's algorithm to find the Eulerian path
    while stack:
        vertex = stack[-1]
        # Find the index of the current vertex
        i = nodes.index(vertex)
        # Find the indices of outgoing edges
        out_indices = [j for j, val in enumerate(edge_matrix[i]) if val == 1]
        if out_indices:
            next_vertex = nodes[out_indices[0]]
            #print(next_vertex )
            # Update edge matrix
            edge_matrix[i][out_indices[0]] = 0
            # Push next vertex to stack
            stack.append(next_vertex)
        else:  # If there are no outgoing edges, backtrack
            path.append(stack.pop())



    # Check if all edges are visited
    for row in edge_matrix:
        for edge in row:
            if edge != 0 and edge is not None:
                print("Not all edges are visited")
            break

    print(len(path))

    # Since we've stored the path in reverse, reverse it to display correctly
    return path[::-1]
```

[7]:

```
adj_list = defaultdict(list)
for node, neighbors in graph.items():
    adj_list[node].extend(neighbors)
    for neighbor in neighbors:
        adj_list[neighbor]
eulerian_path=find_eulerian_path(adj_list,nodes)
#print("Eulerian Path:", eulerian_path)
```

```
Not all edges are visited
783
```

[8]:

```
def assemble_sequence_from_kmers(eulerian_path, k):
    # Initialize the sequence with the first kmer
    sequence = eulerian_path[0]
    # Iterate through the remaining kmers
    sequence = eulerian_path[0]
    # Iterate through the remaining kmers
    for kmer in eulerian_path[1:]:
            sequence += kmer[-1]
    return sequence


# Assemble the sequence from kmers
assembled_sequence = assemble_sequence_from_kmers(eulerian_path, 10)
print("Assembled Sequence:", assembled_sequence)
print("lenth of Assembled Sequence is", len(assembled_sequence))
```

```
Assembled Sequence: ATGCATCGCGGCCCAAGCGATTTCCAAATAACACACATTCATCTAGCAGTGAACTGTCTG
TAGCCAGCATCCTCAGTGTATGTATGGTTCAAGGAGTGATATGGCCCATCTCGGACTAATCTCTTGCTGGCACCTGCTAT
ATTACACTACTGCTCTGCTTAGAACCGCACGTTGACTATCGATTGCTCAACGGATGGTCCGTAACTGACCCAACCTGCGG
```

```
GGAGGACAACTAAGCTGTATTGATGCGCGCCGCACATGCAGACTACTTTGCAAATAGCGCGCGAGATAAGGCCGCAAGAC
AGATCGGGCTGAATTCTGAAAAGTGGATCTTGCATTTATATACGTCAGCAGGTCCTGCATAGGTGAGATAATATTGTCAT
AATTTCGAAAACCTGGCACGGACGAGTCGCTAAACTAGTTTAGTTGCGCACAGGAGACGCCTAGACATAGAGAACCCTGC
CAAAGGGTTCTTCGAAGACGCATTCTTTGGATAACTCGAAGCGACGCTTCTTCGGAAGTAGGGCGGGCACGTTCGACCCT
ACCATCCAAATTTGCTGAGGCGCCATGTTTATGAAGACCCCAGGTGTCCACATCAGTAATGGAACCCCACAATCCTTTAT
GAACCTAGTTTGTCTGACGGTGCGTGGCCTCCCTCTACCATCGGAGCCTGTGGCACATCGGTGGTCGGCTGGTGATGCAG
GTGTAAGATGAGCCGAAGACGGTGCGTCACTCCGGGGTCGGGGGCAATGACCGGCATTTTGGGGAACTCAGCTACTACAG
TTGCCTACCTTA
lenth of Assembled Sequence is 792
```

As we can observe, the find_eulerian_path function alone is unable to provide a complete sequence covering the entirety of the sample sequence. Therefore, we are exploring options to enhance our code in order to generate multiple contigs that collectively cover the entire sample sequence.

```
[9]: def contig(start_vertex, nodes, edge_matrix):
         """
         This function finds contigs starting from a given vertex in a directed␣
      ↪graph.
         """
         # List to store the path
         path = []
         stack = []
         stack.append(start_vertex)

         # Hierholzer's algorithm to find the contigs
         while stack:
             vertex = stack[-1]
             # Find the index of the current vertex
             i = nodes.index(vertex)
             # Find the indices of outgoing edges
             out_indices = [j for j, val in enumerate(edge_matrix[i]) if val == 1]
             if out_indices:
                 next_vertex = nodes[out_indices[0]]

                 # Update edge matrix, 0 means visited
                 edge_matrix[i][out_indices[0]] = 0
                 # Push next vertex to stack
                 stack.append(next_vertex)
             else:  # If there are no outgoing edges, backtrack
                 path.append(stack.pop())

         return path[::-1]


     def find_all_contigs(adj, nodes, k):
         """
         This function finds all contigs from a given directed graph and k-mer␣
      ↪length.
```

```
    """
    nodes = list(nodes)

    # Calculate in-degree and out-degree for each vertex
    in_degree = {i: 0 for i in nodes}
    out_degree = {i: 0 for i in nodes}
    edge_matrix = [[None] * len(nodes) for _ in range(len(nodes))]
    contigs = []

    for node in nodes:
        for neighbor in adj[node]:
            out_degree[node] += 1
            in_degree[neighbor] += 1
            i = nodes.index(node) # Find the index of the node
            j = nodes.index(neighbor) # Find the index of the neighbor node
            edge_matrix[i][j] = 1    # If there is an edge, update the␣
↪corresponding cell to 1

    # Start from a vertex with non-zero out-degree
    for vertex in nodes:
        if out_degree[vertex] != 0 and in_degree[vertex] == 0:
            start_vertex = vertex
            break

    path = contig(start_vertex, nodes, edge_matrix)
    contigs.append(assemble_sequence_from_kmers(path, k))
    # Check if all edges are visited
    for i, row in enumerate(edge_matrix):
        for j, entry in enumerate(row):
            if entry != 0 and entry is not None:
                start_vertex = nodes[i]
                path = contig(start_vertex, nodes, edge_matrix)
                contigs.append(assemble_sequence_from_kmers(path, k))
    return contigs
```

[10]:
```
adj_list = defaultdict(list)
for node, neighbors in graph.items():
    adj_list[node].extend(neighbors)
    for neighbor in neighbors:
        adj_list[neighbor]
contigs=find_all_contigs(adj_list,nodes,10)
```

[11]:
```
print(len(contigs))
```

```
15
```

Part E

[12]:
```python
# Reference: https://biopython.org/docs/1.75/api/Bio.Seq.html

from Bio.Seq import Seq
from Bio import pairwise2

def find_overlap(sequence1, sequence2):
    # Create Seq objects from the input sequences
    seq1 = Seq(sequence1)
    seq2 = Seq(sequence2)

    # Find the alignment between the two sequences
    alignments = pairwise2.align.localms(seq1, seq2, 1, -1, -1, -1)

    # Extract start and end points of the overlap from the first alignment
    alignment = alignments[0]
    start = alignment.start
    end = alignment.end

    return start, end
```

```
/Users/linyu/opt/anaconda3/lib/python3.9/site-packages/Bio/pairwise2.py:278:
BiopythonDeprecationWarning: Bio.pairwise2 has been deprecated, and we intend to
remove it in a future release of Biopython. As an alternative, please consider
using Bio.Align.PairwiseAligner as a replacement, and contact the Biopython
developers if you still need the Bio.pairwise2 module.
  warnings.warn(
```

[13]:
```python
starts=[]
ends=[]
for i, contig in enumerate(contigs):
    start, end = find_overlap(contig, sample_geno)
    starts.append(start)
    ends.append(end)
    print(f"The {i+1}th contig overlaps with the sample genome:")
    print("Start:", start)
    print("End:", end)
```

```
The 1th contig overlaps with the sample genome:
Start: 55
End: 847
The 2th contig overlaps with the sample genome:
Start: 959
End: 1000
The 3th contig overlaps with the sample genome:
Start: 37
End: 64
The 4th contig overlaps with the sample genome:
Start: 876
```

```
End: 969
The 5th contig overlaps with the sample genome:
Start: 863
End: 886
The 6th contig overlaps with the sample genome:
Start: 12
End: 47
The 7th contig overlaps with the sample genome:
Start: 855
End: 873
The 8th contig overlaps with the sample genome:
Start: 850
End: 865
The 9th contig overlaps with the sample genome:
Start: 844
End: 860
The 10th contig overlaps with the sample genome:
Start: 6
End: 22
The 11th contig overlaps with the sample genome:
Start: 841
End: 854
The 12th contig overlaps with the sample genome:
Start: 839
End: 851
The 13th contig overlaps with the sample genome:
Start: 5
End: 16
The 14th contig overlaps with the sample genome:
Start: 838
End: 849
The 15th contig overlaps with the sample genome:
Start: 4
End: 15
```

[14]:
```python
print("Our contigs cover from", min(starts), "to", max(ends), "bps of the␣
 ↪sample genome.")
```

```
Our contigs cover from 4 to 1000 bps of the sample genome.
```