JHU_Bioinformatics_Module3_Homework_Practical-genomic_assembly_with_overlap_graphs (1)

May 28, 2025

```
[ ]: Module 3: Edit Distance, Alignment & Sequence Assembly
Programming Homework – Genomic Data Science Specialization (Johns Hopkins)
Course: Algorithms for DNA Sequencing
Specialization: Bioinformatics – Genomic Data Science
Author: Julian Borges
Module: 3 – Edit Distance, Alignment, and Overlap Graphs

This notebook documents the implementation of algorithms and exercises related␣
 ↪to:

Edit Distance (Dynamic Programming)
Global and Local Alignment
Read Mapping and Approximate Matching
Sequence Assembly using Overlap Graphs
Each section includes:

Practical exercises
Annotated Python code
Validated outputs
Reflection prompts to reinforce understanding
All tasks are based on real-world genome sequencing data scenarios.
```

1 Genome Assembly Using Overlap Graphs

This section introduces the concept of overlap graphs and how they are used in genome assembly.

Overlaps are essential for assembling genomes from short reads. They act as the 'glue' that helps us piece together the original sequence.

1.1 Directed Graphs

We use a **directed graph** to represent overlaps. In this graph: - Each **node** is a sequencing read. - A **directed edge** from node A to node B indicates that a **suffix** of A overlaps with a **prefix** of B.

To be meaningful, we set a threshold on overlap length (e.g., at least 4 bases and exact match).

1.2 Constructing the Overlap Graph

We can construct an overlap graph from all k-mers in a sequence. For example, given a synthetic genome sequence, extract all 6-mers and link those with overlapping suffixes/prefixes.

This graph can help us trace a path that reconstructs the genome.

[ ]:
```
# Simple code to find suffix-prefix overlaps of length >= threshold
def overlap(a, b, min_length=4):
    start = 0  # start all the way at the left
    while True:
        start = a.find(b[:min_length], start)
        if start == -1:
            return 0
        if b.startswith(a[start:]):
            return len(a) - start
        start += 1

reads = ['GTACGT', 'TACGTA', 'ACGTAC', 'CGTACG', 'GTACGA']
edges = []

for a in reads:
    for b in reads:
        if a != b:
            olen = overlap(a, b, min_length=4)
            if olen > 0:
                edges.append((a, b, olen))

edges
```

1.3 Path through the Graph and Genome Reconstruction

Walking through this graph (following the maximum overlap edges) reconstructs the genome by joining reads using the overlap. This is how we go from short reads to a longer contiguous sequence.

[ ]:
```
# Reconstruct genome from graph by greedy path extension
from collections import defaultdict

# Build graph
graph = defaultdict(list)
for a, b, olen in edges:
    graph[a].append((b, olen))

# Choose starting node (one not present as a destination)
destinations = set(b for _, b, _ in edges)
start_node = [node for node in reads if node not in destinations][0]

# Reconstruct sequence
sequence = start_node
```

```
current = start_node

while graph[current]:
    next_node, olen = max(graph[current], key=lambda x: x[1])
    sequence += next_node[olen:]
    current = next_node

sequence
```

1.4 Practical: Implementing an Overlap Function

In this practical section, we'll implement a Python function that computes the length of the longest suffix of one string (`a`) that matches a prefix of another string (`b`), where the overlap is at least a specified minimum length. This will be a core utility in genome assembly using overlap graphs.

[ ]:
```
def overlap(a, b, min_length=3):
    """Return length of the longest suffix of 'a' matching
    a prefix of 'b' that is at least 'min_length' characters long.
    If no such overlap exists, return 0."""
    start = 0  # start all the way at the left
    while True:
        start = a.find(b[:min_length], start)  # look for b's prefix in a
        if start == -1:
            return 0  # no more occurrences to the right
        # found occurrence; check for full suffix/prefix match
        if b.startswith(a[start:]):
            return len(a) - start
        start += 1
```

[ ]:
```
# Test the overlap function
print(overlap("CGTACG", "CGTACGT", 3))  # Expected: 6
print(overlap("TTACG", "ACGTT", 3))     # Expected: 3
print(overlap("GATTACA", "TACAG", 4))   # Expected: 4
print(overlap("GATTACA", "AGGT", 3))    # Expected: 0
```

[ ]:
```
# Global Alignment using Scoring Matrix
This section implements global alignment using a scoring matrix. Unlike edit␣
↪distance, we assign different penalties for mismatches and gaps based on␣
↪biological context.
```

[ ]:
```
# Define the alphabet and scoring matrix
alphabet = ['A', 'C', 'G', 'T']
score = [
    [0, 4, 2, 4, 8],  # A
    [4, 0, 4, 2, 8],  # C
    [2, 4, 0, 4, 8],  # G
    [4, 2, 4, 0, 8],  # T
    [8, 8, 8, 8, 8]   # gap penalties (last row and col)
```

```
]
```

[ ]:
```
def globalAlignment(x, y):
    D = []
    for i in range(len(x) + 1):
        D.append([0] * (len(y) + 1))

    # Initialize first column
    for i in range(1, len(x) + 1):
        D[i][0] = D[i - 1][0] + score[alphabet.index(x[i - 1])][4]

    # Initialize first row
    for j in range(1, len(y) + 1):
        D[0][j] = D[0][j - 1] + score[4][alphabet.index(y[j - 1])]

    # Fill the matrix
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            distHor = D[i][j - 1] + score[4][alphabet.index(y[j - 1])]
            distVer = D[i - 1][j] + score[alphabet.index(x[i - 1])][4]
            if x[i - 1] == y[j - 1]:
                distDiag = D[i - 1][j - 1]
            else:
                distDiag = D[i - 1][j - 1] + score[alphabet.index(x[i -␣
 ↪1])][alphabet.index(y[j - 1])]
            D[i][j] = min(distHor, distVer, distDiag)

    return D[-1][-1]
```

[ ]:
```
# Test case for global alignment
x = "GATTACA"
y = "GCATGCU"
print("Global alignment penalty:", globalAlignment(x, y))
```

[ ]:
```
# Global Alignment using Scoring Matrix
This section implements global alignment using a scoring matrix. Unlike edit␣
 ↪distance, we assign different penalties for mismatches and gaps based on␣
 ↪biological context.
```

[ ]:
```
# Define the alphabet and scoring matrix
alphabet = ['A', 'C', 'G', 'T']
score = [
    [0, 4, 2, 4, 8],  # A
    [4, 0, 4, 2, 8],  # C
    [2, 4, 0, 4, 8],  # G
    [4, 2, 4, 0, 8],  # T
    [8, 8, 8, 8, 8]   # gap penalties (last row and col)
```

```
]
```

[ ]:
```
def globalAlignment(x, y):
    D = []
    for i in range(len(x) + 1):
        D.append([0] * (len(y) + 1))

    # Initialize first column
    for i in range(1, len(x) + 1):
        D[i][0] = D[i - 1][0] + score[alphabet.index(x[i - 1])][4]

    # Initialize first row
    for j in range(1, len(y) + 1):
        D[0][j] = D[0][j - 1] + score[4][alphabet.index(y[j - 1])]

    # Fill the matrix
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            distHor = D[i][j - 1] + score[4][alphabet.index(y[j - 1])]
            distVer = D[i - 1][j] + score[alphabet.index(x[i - 1])][4]
            if x[i - 1] == y[j - 1]:
                distDiag = D[i - 1][j - 1]
            else:
                distDiag = D[i - 1][j - 1] + score[alphabet.index(x[i -␣
 ↪1])][alphabet.index(y[j - 1])]
            D[i][j] = min(distHor, distVer, distDiag)

    return D[-1][-1]
```

[ ]:
```
# Test case for global alignment
x = "GATTACA"
y = "GCATGCU"
print("Global alignment penalty:", globalAlignment(x, y))
```

[ ]:
```
# Genome Assembly Using Overlap Graphs

This section introduces the concept of overlap graphs and how they are used in␣
 ↪genome assembly.

Overlaps are essential for assembling genomes from short reads. They act as the␣
 ↪'glue' that helps us piece together the original sequence.
```

[ ]:
```
## Directed Graphs

We use a **directed graph** to represent overlaps. In this graph:
- Each **node** is a sequencing read.
```

```
- A **directed edge** from node A to node B indicates that a **suffix** of A overlaps with a **prefix** of B.

To be meaningful, we set a threshold on overlap length (e.g., at least 4 bases and exact match).
```

[ ]:
```
## Constructing the Overlap Graph

We can construct an overlap graph from all k-mers in a sequence.
For example, given a synthetic genome sequence, extract all 6-mers and link those with overlapping suffixes/prefixes.

This graph can help us trace a path that reconstructs the genome.
```

[ ]:
```
# Simple code to find suffix-prefix overlaps of length >= threshold
def overlap(a, b, min_length=4):
    start = 0  # start all the way at the left
    while True:
        start = a.find(b[:min_length], start)
        if start == -1:
            return 0
        if b.startswith(a[start:]):
            return len(a) - start
        start += 1

reads = ['GTACGT', 'TACGTA', 'ACGTAC', 'CGTACG', 'GTACGA']
edges = []

for a in reads:
    for b in reads:
        if a != b:
            olen = overlap(a, b, min_length=4)
            if olen > 0:
                edges.append((a, b, olen))

edges
```

[ ]:
```
# Simple code to find suffix-prefix overlaps of length >= threshold
def overlap(a, b, min_length=4):
    start = 0  # start all the way at the left
    while True:
        start = a.find(b[:min_length], start)
        if start == -1:
            return 0
        if b.startswith(a[start:]):
            return len(a) - start
        start += 1
```

```
reads = ['GTACGT', 'TACGTA', 'ACGTAC', 'CGTACG', 'GTACGA']
edges = []

for a in reads:
    for b in reads:
        if a != b:
            olen = overlap(a, b, min_length=4)
            if olen > 0:
                edges.append((a, b, olen))

edges
```

[ ]:
```
## Path through the Graph and Genome Reconstruction

Walking through this graph (following the maximum overlap edges) reconstructs␣
↪the genome by joining reads using the overlap.
This is how we go from short reads to a longer contiguous sequence.
```

[ ]:
```
# Reconstruct genome from graph by greedy path extension
from collections import defaultdict

# Build graph
graph = defaultdict(list)
for a, b, olen in edges:
    graph[a].append((b, olen))

# Choose starting node (one not present as a destination)
destinations = set(b for _, b, _ in edges)
start_node = [node for node in reads if node not in destinations][0]

# Reconstruct sequence
sequence = start_node
current = start_node

while graph[current]:
    next_node, olen = max(graph[current], key=lambda x: x[1])
    sequence += next_node[olen:]
    current = next_node

sequence
```

[ ]:
```
## Practical: Implementing an Overlap Function
```

In this practical section, we'll implement a Python function that computes the length of the longest suffix of one string (`a`) that matches a prefix of another string (`b`), where the overlap is at least a specified minimum length. This will be a core utility in genome assembly using overlap graphs.

[ ]:
```python
def overlap(a, b, min_length=3):
    """Return length of the longest suffix of 'a' matching
    a prefix of 'b' that is at least 'min_length' characters long.
    If no such overlap exists, return 0."""
    start = 0  # start all the way at the left
    while True:
        start = a.find(b[:min_length], start)  # look for b's prefix in a
        if start == -1:
            return 0  # no more occurrences to the right
        # found occurrence; check for full suffix/prefix match
        if b.startswith(a[start:]):
            return len(a) - start
        start += 1
```

[ ]:
```python
# Test the overlap function
print(overlap("CGTACG", "CGTACGT", 3))  # Expected: 6
print(overlap("TTACG", "ACGTT", 3))     # Expected: 3
print(overlap("GATTACA", "TACAG", 4))   # Expected: 4
print(overlap("GATTACA", "AGGT", 3))    # Expected: 0
```

[ ]:
```python
# Define the alphabet and scoring matrix
alphabet = ['A', 'C', 'G', 'T']
score = [
    [0, 4, 2, 4, 8],  # A
    [4, 0, 4, 2, 8],  # C
    [2, 4, 0, 4, 8],  # G
    [4, 2, 4, 0, 8],  # T
    [8, 8, 8, 8, 8]   # gap penalties (last row and col)
]
```

[ ]:
```python
def globalAlignment(x, y):
    D = []
    for i in range(len(x) + 1):
        D.append([0] * (len(y) + 1))

    # Initialize first column
    for i in range(1, len(x) + 1):
        D[i][0] = D[i - 1][0] + score[alphabet.index(x[i - 1])][4]

    # Initialize first row
    for j in range(1, len(y) + 1):
        D[0][j] = D[0][j - 1] + score[4][alphabet.index(y[j - 1])]
```

```
    # Fill the matrix
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            distHor = D[i][j - 1] + score[4][alphabet.index(y[j - 1])]
            distVer = D[i - 1][j] + score[alphabet.index(x[i - 1])][4]
            if x[i - 1] == y[j - 1]:
                distDiag = D[i - 1][j - 1]
            else:
                distDiag = D[i - 1][j - 1] + score[alphabet.index(x[i - 1])][alphabet.index(y[j - 1])]
            D[i][j] = min(distHor, distVer, distDiag)

    return D[-1][-1]
```

[ ]:
```
# Test case for global alignment
x = "GATTACA"
y = "GCATGCU"
print("Global alignment penalty:", globalAlignment(x, y))
```

[ ]:
```
# Global Alignment using Scoring Matrix
This section implements global alignment using a scoring matrix. Unlike edit distance, we assign different penalties for mismatches and gaps based on biological context.
```

[ ]:
```
# Define the alphabet and scoring matrix
alphabet = ['A', 'C', 'G', 'T']
score = [
    [0, 4, 2, 4, 8],  # A
    [4, 0, 4, 2, 8],  # C
    [2, 4, 0, 4, 8],  # G
    [4, 2, 4, 0, 8],  # T
    [8, 8, 8, 8, 8]   # gap penalties (last row and col)
]
```

[ ]:
```
def globalAlignment(x, y):
    D = []
    for i in range(len(x) + 1):
        D.append([0] * (len(y) + 1))

    # Initialize first column
    for i in range(1, len(x) + 1):
        D[i][0] = D[i - 1][0] + score[alphabet.index(x[i - 1])][4]

    # Initialize first row
    for j in range(1, len(y) + 1):
        D[0][j] = D[0][j - 1] + score[4][alphabet.index(y[j - 1])]
```

```
    # Fill the matrix
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            distHor = D[i][j - 1] + score[4][alphabet.index(y[j - 1])]
            distVer = D[i - 1][j] + score[alphabet.index(x[i - 1])][4]
            if x[i - 1] == y[j - 1]:
                distDiag = D[i - 1][j - 1]
            else:
                distDiag = D[i - 1][j - 1] + score[alphabet.index(x[i - 1])][alphabet.index(y[j - 1])]
            D[i][j] = min(distHor, distVer, distDiag)

    return D[-1][-1]
```

[ ]:
```
# Test case for global alignment
x = "GATTACA"
y = "GCATGCU"
print("Global alignment penalty:", globalAlignment(x, y))
```

[ ]:
```
# Genome Assembly Using Overlap Graphs

This section introduces the concept of overlap graphs and how they are used in genome assembly.

Overlaps are essential for assembling genomes from short reads. They act as the 'glue' that helps us piece together the original sequence.
```

[ ]:
```
## Directed Graphs

We use a **directed graph** to represent overlaps. In this graph:
- Each **node** is a sequencing read.
- A **directed edge** from node A to node B indicates that a **suffix** of A overlaps with a **prefix** of B.

To be meaningful, we set a threshold on overlap length (e.g., at least 4 bases and exact match).
```

[ ]:
```
## Constructing the Overlap Graph

We can construct an overlap graph from all k-mers in a sequence.
For example, given a synthetic genome sequence, extract all 6-mers and link those with overlapping suffixes/prefixes.

This graph can help us trace a path that reconstructs the genome.
```

[ ]:
```python
# Simple code to find suffix-prefix overlaps of length >= threshold
def overlap(a, b, min_length=4):
    start = 0  # start all the way at the left
    while True:
        start = a.find(b[:min_length], start)
        if start == -1:
            return 0
        if b.startswith(a[start:]):
            return len(a) - start
        start += 1

reads = ['GTACGT', 'TACGTA', 'ACGTAC', 'CGTACG', 'GTACGA']
edges = []

for a in reads:
    for b in reads:
        if a != b:
            olen = overlap(a, b, min_length=4)
            if olen > 0:
                edges.append((a, b, olen))

edges
```

[ ]:
```python
## Path through the Graph and Genome Reconstruction

Walking through this graph (following the maximum overlap edges) reconstructs␣
 ↪the genome by joining reads using the overlap.
This is how we go from short reads to a longer contiguous sequence.
```

[ ]:
```python
# Reconstruct genome from graph by greedy path extension
from collections import defaultdict

# Build graph
graph = defaultdict(list)
for a, b, olen in edges:
    graph[a].append((b, olen))

# Choose starting node (one not present as a destination)
destinations = set(b for _, b, _ in edges)
start_node = [node for node in reads if node not in destinations][0]

# Reconstruct sequence
sequence = start_node
current = start_node

while graph[current]:
    next_node, olen = max(graph[current], key=lambda x: x[1])
```

```
    sequence += next_node[olen:]
    current = next_node

sequence
```

[ ]:
```
## Practical: Implementing an Overlap Function

In this practical section, we'll implement a Python function that computes the␣
↪length of the longest suffix of one string (`a`) that matches a prefix of␣
↪another string (`b`), where the overlap is at least a specified minimum␣
↪length. This will be a core utility in genome assembly using overlap graphs.
```

[ ]:
```
def overlap(a, b, min_length=3):
    """Return length of the longest suffix of 'a' matching
    a prefix of 'b' that is at least 'min_length' characters long.
    If no such overlap exists, return 0."""
    start = 0  # start all the way at the left
    while True:
        start = a.find(b[:min_length], start)  # look for b's prefix in a
        if start == -1:
            return 0  # no more occurrences to the right
        # found occurrence; check for full suffix/prefix match
        if b.startswith(a[start:]):
            return len(a) - start
        start += 1
```

[ ]:
```
# Test the overlap function
print(overlap("CGTACG", "CGTACGT", 3))  # Expected: 6
print(overlap("TTACG", "ACGTT", 3))     # Expected: 3
print(overlap("GATTACA", "TACAG", 4))   # Expected: 4
print(overlap("GATTACA", "AGGT", 3))    # Expected: 0
```