Penerapan Algoritma *Branch and Bound* dalam *DNA Sequencing Assembly*

Randy Verdian - 13522067
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522067@std.stei.itb.ac.id

***Abstrak*—DNA Sequencing Assembly merupakan permasalahan dalam bioinformatika, yang melibatkan penyusunan kembali fragmen DNA pendek menjadi satu urutan panjang yang representatif. Dalam makalah ini, saya membuat pendekatan berbasis algoritma Branch and Bound untuk perakitan sekuens DNA. Metode ini secara efektif mengurangi ruang pencarian dengan memangkas cabang-cabang yang tidak mengarah pada solusi optimal, sehingga mempercepat proses perakitan. Dengan memanfaatkan konsep overlap maksimum antara fragmen DNA untuk membentuk superstring terpendek yang menggabungkan semua fragmen input. Pendekatan ini menawarkan solusi yang lebih cepat dan menghasilkan optimal untuk perakitan sekuens DNA, yang dapat memberikan kontribusi dalam penelitian genomik dan aplikasi bioteknologi.**

Kata kunci—DNA Sequencing Assembly; Branch and Bound; genomik; bioinformatika

I. Pendahuluan

DNA Sequencing Assembly adalah sebuah bidang penting dalam bioinformatika yang berperan besar dalam penelitian genomik dan aplikasi bioteknologi. Tujuannya adalah menyusun kembali fragmen-fragmen DNA pendek yang diperoleh dari proses sequencing menjadi satu urutan DNA panjang yang representatif. Tantangan utama dalam DNA Sequencing Assembly adalah mengatasi kompleksitas kombinatorial yang tinggi dalam menentukan urutan fragmen yang tepat, yang dapat mengakibatkan solusi yang tidak efisien dan memerlukan waktu komputasi yang signifikan.

Pendekatan tradisional dalam DNA Sequencing Assembly seperti algoritma Greedy, Dynamic Programming, dan Heuristik, meskipun telah memberikan kontribusi signifikan, seringkali tidak mampu menangani kompleksitas besar dengan efisiensi yang memadai. Dalam konteks ini, algoritma Branch and Bound muncul sebagai solusi yang potensial untuk meningkatkan efisiensi dan akurasi dalam perakitan sekuens DNA.

Algoritma Branch and Bound adalah metode pencarian yang memanfaatkan strategi pembagian ruang pencarian menjadi sub-problem yang lebih kecil (branching) dan memangkas (bounding) cabang-cabang yang tidak mengarah pada solusi optimal. Metode ini terkenal karena kemampuannya dalam mengurangi ruang pencarian secara drastis, yang pada gilirannya mempercepat proses pencarian solusi optimal. Dalam perakitan sekuens DNA, algoritma Branch and Bound dapat diterapkan dengan cara memaksimalkan overlap antara fragmen-fragmen DNA untuk membentuk superstring terpendek yang menggabungkan semua fragmen input.

Makalah ini bertujuan untuk mengkaji penerapan algoritma Branch and Bound dalam konteks DNA Sequencing Assembly. Dengan menggunakan algoritma ini, diharapkan dapat diperoleh solusi perakitan sekuens DNA yang lebih cepat dan optimal dibandingkan metode-metode konvensional. Lebih lanjut, pendekatan ini diharapkan dapat memberikan kontribusi yang signifikan dalam penelitian genomik dan aplikasi bioteknologi dengan menyediakan alat yang efisien untuk perakitan sekuens DNA.

II. Landasan Teori

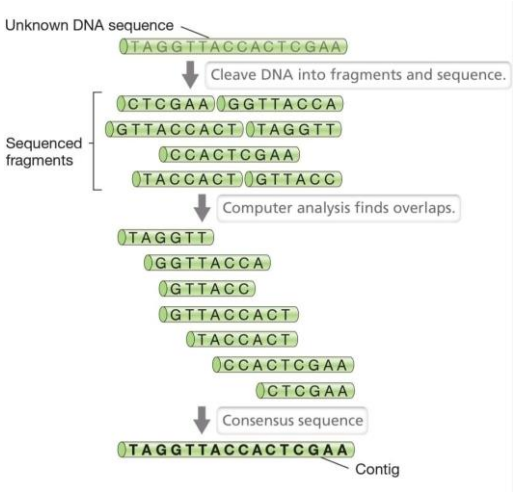
A. Struktur DNA

Struktur DNA terdiri dari dua untai polinukleotida yang berpilin membentuk double helix. Setiap untai tersusun dari unit berulang yang disebut nukleotida, yang masing-masing terdiri dari tiga komponen: gula deoksiribosa, gugus fosfat, dan salah satu dari empat basa nitrogen yaitu adenin (A), timin (T), sitosin (C), dan guanin (G). Basa-basa nitrogen ini berpasangan secara spesifik melalui ikatan hidrogen, di mana adenin selalu berpasangan dengan timin dan sitosin dengan guanin, membentuk pasangan basa komplementer. Ikatan antara nukleotida membentuk tulang punggung heliks, dengan basa nitrogen yang menghadap ke dalam, memungkinkan interaksi yang menjaga kestabilan struktur heliks ganda.

B. DNA Sequencing Assembly

DNA Sequencing Assembly adalah proses untuk menyusun kembali fragmen-fragmen DNA pendek yang diperoleh dari teknik sequencing menjadi satu urutan DNA panjang yang representatif. Teknik sequencing seperti shotgun sequencing menghasilkan banyak fragmen DNA pendek yang saling tumpang tindih, yang kemudian perlu digabungkan menjadi satu sekuens utuh. Tantangan utama dalam DNA Sequencing Assembly adalah menentukan urutan yang benar

dari fragmen-fragmen ini, yang sering kali melibatkan analisis kombinatorial yang kompleks.



Gambar 1. Perakitan sekuens DNA
Sumber: https://www.linkedin.com/posts/hanaa-fayed-1275a065_computer-assembly-of-dna-sequence-most-activity-7093239381058084866-uYLG?utm_source=combined_share_message&utm_medium=member_desktop

1. Overlap-Layout-Consensus (OLC) Model

Model OLC adalah salah satu pendekatan utama dalam DNA Sequencing Assembly. Model ini terdiri dari tiga tahap utama:

- Overlap: Mengidentifikasi fragmen-fragmen yang memiliki tumpang tindih signifikan.
- Layout: Menyusun urutan fragmen berdasarkan tumpang tindih yang teridentifikasi.
- Consensus: Menggabungkan fragmen-fragmen tersebut menjadi satu urutan konsensus.

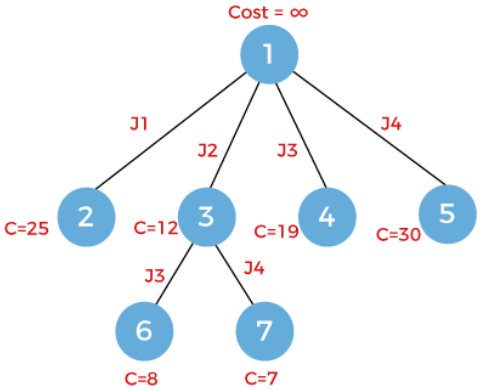
2. De Bruijn Graph

Pendekatan lain yang populer adalah penggunaan de Bruijn graph, di mana fragmen-fragmen DNA dipecah menjadi k-mers (subsekuens sepanjang k), dan node dalam graph mewakili k-mers tersebut. Edge antara node menunjukkan k-mers yang tumpang tindih.

C. Algoritma Branch and Bound

Algoritma branch and bound digunakan untuk menemukan solusi optimal dari masalah optimasi yang bersifat kombinatorik, diskrit, dan matematis umum. Metode ini mengeksplorasi seluruh ruang pencarian untuk mengidentifikasi kandidat solusi secara bertahap. Teknik ini cocok untuk digunakan dalam masalah optimasi diskrit seperti pemrograman integer 0-1 dan masalah aliran jaringan, serta masalah optimasi kombinatorik seperti Boolean Satisfiability dan Integer Linear Programming.

Algoritma ini bekerja dengan membuat pohon ruang keadaan dan menggunakan fungsi batas untuk mengevaluasi solusi kandidat. Terdapat tiga teknik pencarian dalam branch and bound: Least Cost Search (LC), Breadth First Search (BFS), dan Depth First Search (DFS). LC search menggunakan fungsi biaya heuristik untuk memilih node dengan nilai terendah. BFS menggunakan antrean untuk memproses node dalam urutan masuk pertama keluar pertama, sedangkan DFS menggunakan tumpukan untuk memproses node dalam urutan masuk terakhir keluar pertama.



Gambar 2. Algoritma Branch and Bound
Sumber: https://static.javatpoint.com/tutorial/daa/images/branch-and-bound19.png

Branch and bound mencakup berbagai jenis teknik pencarian seperti FIFO, LIFO, dan Least Cost Branch and Bound. Metode FIFO mengeksplorasi node berdasarkan urutan antrean, metode LIFO menggunakan tumpukan, sementara Least Cost Branch and Bound mengeksplorasi node berdasarkan fungsi biaya, menghentikan eksplorasi pada node yang tidak memberikan solusi lebih baik dari solusi terbaik yang sudah ditemukan.

Keuntungan dari algoritma ini termasuk efisiensi waktu karena tidak perlu mengeksplorasi semua node, dan kemampuan menemukan solusi optimal dalam waktu minimal pada masalah kecil. Namun, kelemahannya adalah waktu eksekusi yang lama dan jumlah node yang besar dalam kasus terburuk. Algoritma branch and bound dapat diterapkan pada berbagai masalah kombinatorik seperti masalah knapsack 0/1, masalah teka-teki 8, masalah penugasan pekerjaan, masalah N Queen, dan masalah Traveling Salesman.

III. PENYELESAIAN MASALAH

A. Pendekatan Solusi

Dalam DNA Sequencing Assembly, terdapat input berupa fragmen-fragmen DNA yang akan disusun, dan outputnya adalah sequence DNA dengan panjang yang terpendek atau superstring terpendek, dengan kata lain jumlah overlaping yang terjadi maksimum.

Untuk menemukan solusinya, akan dilakukan pendekatan dengan salah satu kasus uji, yaitu sebagai berikut:

Input fragmen DNA:

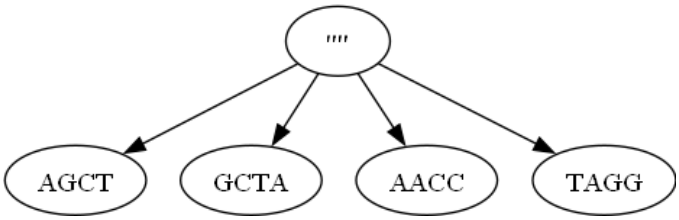
AGCT

GCTA

AACC

TAGG

Akan dibuat graphnya, misalkan *root* nya adalah string kosong



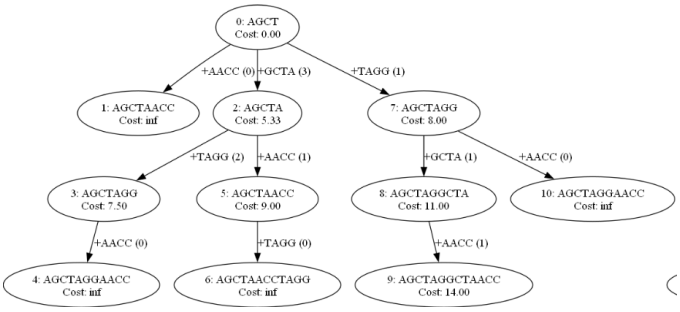
Kemudian, algoritma memulai pencarian dari setiap fragmen sebagai titik awal, yang mana setiap fragmen tersebut diinisialisasi *cost* nya adalah nol. Dalam algoritma branch and bound, awalnya *branching* mulai dibuat dimana Algoritma membangun cabang baru untuk setiap kemungkinan penggabungan fragmen yang tersisa dengan fragmen terakhir dalam jalur saat ini. Setiap cabang baru mewakili keadaan baru dari superstring yang dihasilkan dengan menambahkan fragmen baru. Sebelum membuat cabang baru, algoritma memeriksa apakah panjang superstring sementara saat ini sudah lebih panjang dari superstring terpendek yang ditemukan sejauh ini. Jika ya, cabang tersebut dipangkas atau di-*purning* karena tidak mungkin menghasilkan solusi yang lebih baik. Jika dua fragmen tidak memiliki tumpang tindih (biaya -inf), cabang tersebut juga dipangkas. Misalkan fungsi pembatasnya adalah *cost_function* dimana fungsi tersebut untuk mengevaluasi biaya dari superstring sementara yang sedang dibentuk. Biaya ini digunakan untuk memutuskan jalur mana yang lebih menjanjikan untuk dilanjutkan. Misalkan fungsi pembatas (*cost_function*) dinyatakan dengan

$$c(n) = f(n) + \frac{1}{g(n)}$$

Dimana c(n) adalah *cost* pada node itu, f(n) menyatakan panjang string yang telah dibentuk pada node tersebut, dan g(n) adalah banyaknya overlap dengan fragmen yang digabungkan sebelum string pada node itu terbentuk. Sehingga jika g(n) = 0 atau dengan kata lain, tidak ada overlap yang terjadi, maka c(n) = f(n) + 1/0 = inf, sehingga node ini akan dihentikan dan tidak akan dilakukan percabangan dari node tersebut. Fungsi ini bertujuan untuk menggabungkan dua aspek penting dalam satu metrik biaya yaitu panjang superstring dimana secara umum, kita menginginkan superstring yang sesingkat mungkin. Dan juga overlapping, kita menginginkan tumpang tindih yang sebesar mungkin antara fragmen, karena tumpang tindih yang lebih besar menunjukkan kecocokan yang lebih baik.

Dengan cara ini, algoritma branch and bound dapat menggunakan fungsi biaya ini untuk mengevaluasi dan membandingkan solusi parsial yang berbeda, memilih solusi yang meminimalkan panjang superstring sambil memaksimalkan tumpang tindih antara fragmen.

Kembali ke graph yang sudah dibuat sebelumnya yang mengekspansi AGCT, GCTA, AACC, dan TAGG dengan inisialisasi *cost* bernilai nol. Maka, kita mulai mengekspansi dari node paling kiri yaitu AGCT,



Dari node AGCT, algoritma mencoba menambahkan setiap fragmen yang tersisa ke superstring sementara. Setiap kemungkinan penggabungan menghasilkan node baru di graf.

- Node 1: Menggabungkan "AGCT" dengan "AACC" tanpa tumpang tindih menghasilkan "AGCTAACC" dengan biaya inf, sehingga ekspansi dari node 1 tidak dilanjutkan, maka melanjutkan ke node yang memiliki *cost* paling kecil saat ini, yaitu pada node 2.

- Node 2: Dari "AGCT", menambahkan "GCTA" dengan tumpang tindih 3 menghasilkan "AGCTA" dengan biaya 5.33.

- Node 3: Dari "AGCTA", menambahkan "TAGG" dengan tumpang tindih 2 menghasilkan "AGCTAGG" dengan biaya 7.50.

- Node 4: Dari “AGCTAGG”, menambahkan “AACC” tanpa tumpang tindih menghasilkan “AGCTAGGAACC” dengan biaya inf, maka ekspansi dari node 4 tidak dilanjutkan, maka melanjutkan ke node yang memiliki *cost* paling kecil saat ini, yaitu pada node 5. Namun, pada node 4 adalah *goal* superstring yang sudah mengandung semua fragmen, sehingga dimasukkan ke list solusi superstring.

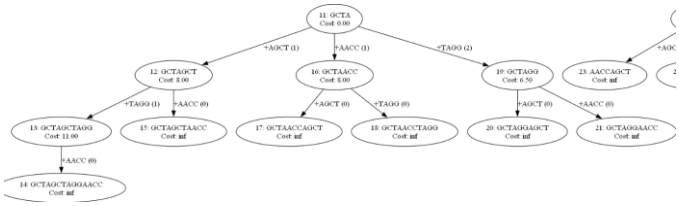
- Node 5: Dari "AGCTA", menambahkan "AACC" tanpa tumpang tindih menghasilkan "AGCTACC" dengan biaya 9.00.

- Node 6: Menggabungkan " AGCTACC " dengan "TAGG" tanpa tumpang tindih menghasilkan "AGCTAACCTAGG" dengan biaya inf, sehingga ekspansi dari node 6 tidak dilanjutkan, maka

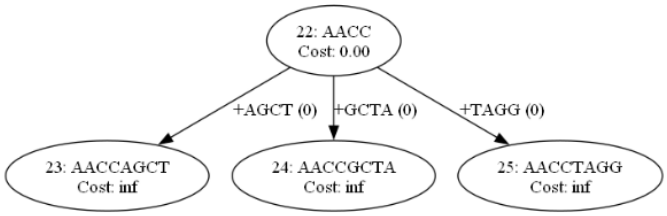
melanjutkan ke node yang memiliki *cost* paling kecil saat ini, yaitu pada node 7. Namun, pada node 6 adalah *goal* superstring yang sudah mengandung semua fragmen, sehingga dimasukkan ke list solusi superstring.

- Node 7: Menggabungkan "AGCT" dengan "TAGG" dengan tumpang tindih 1 menghasilkan "AGCTAGG" dengan biaya 8.00.
- Node 8: Menggabungkan "AGCTAGG" dengan "GCTA" dengan tumpang tindih 1 menghasilkan "AGCTAGGCTA" dengan biaya 11.00.
- Node 9: Menggabungkan "AGCTAGGCTA" dengan "AACC" dengan tumpang tindih 1 menghasilkan " AGCTAGGCTAACC" dengan biaya 14.00, dan node 9 merupakan solusi yang memenuhi karena ini merupakan *goal* superstring.
- Node 10: Dari "AGCTAGG", menambahkan "AACC" tanpa tumpang tindih menghasilkan "AGCTAGGAACC" dengan biaya inf, maka ekspansi dari node 10 tidak dilanjutkan.

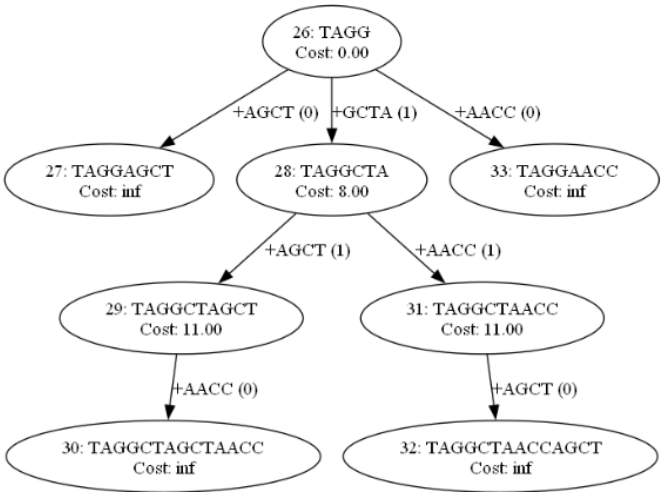
Solusi telah ditemukan pada node 9, namun ada node yang memiliki cost yang lebih kecil yaitu pada *root* node fragmentasi yang lain, yaitu node 11 (GCTA), dengan cara yang sama, diperoleh ekspansi graph sebagai berikut:



Ternyata terdapat satu node yang merupakan *goal* superstring yaitu pada node 14 ("GCTAGCTAGGAACC"), sekarang lanjut ke *root* node berikutnya, yaitu node 22 (AACC):



Ternyata semuanya juga inf dan tidak ada yang sampai *goal* superstring, sekarang lanjut ke *root* node berikutnya, yaitu node 26 (TAGG):



Didapat node 30 ("TAGGCTAGCTAACC") dan node 32 ("TAGGCTAACCAGCT") merupakan *goal* superstring. Dan node selain itu ternyata inf sebelum menemukan *goal* superstring, sehingga semua pencarian dihentikan.

Maka diperoleh list *goal* superstring, yaitu:

Node	Superstring	Panjang
4	AGCTAGGAACC	11
6	AGCTAACCTAGG	12
9	AGCTAGGCTAACC	13
14	GCTAGCTAGGAACC	14
30	TAGGCTAGCTAACC	14
32	TAGGCTAACCAGCT	14

Sehingga, diperoleh superstring dengan panjang terpendek yaitu 11 adalah **AGCTAGGAACC**. Dapat dicek bahwa superstring tersebut sudah terdiri dari fragmen DNA yang diberikan.

B. Implementasi

Dalam pengimplementasian algoritma branch and bound untuk DNA sequencing assembly, dibagi menjadi tiga tahap yaitu overlapping, layout, dan consensus.

1. Overlapping

Pada bagian ini, kita akan mengidentifikasi tumpang tindih antara fragmen-fragmen yang diberikan.

1.1. Membaca Fragmen dari File

```python
def read_fragments_from_file(file_path):
    with open(file_path, 'r') as file:
        fragments = [line.strip() for line in file.readlines()]
    return fragments
```

1.2. Menghitung Panjang Tumpang Tindih antara Dua String

```python
def overlap(a, b, min_length=1):
    start = 0
    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
```

1.3. Membangun Graf Tumpang Tindih

Graf tumpang tindih dibangun untuk menyimpan informasi tentang seberapa besar dua fragmen tumpang tindih satu sama lain:

```python
def build_overlap_graph(fragments):
    overlap_graph = {}
    for a, b in itertools.permutations(fragments, 2):
        olen = overlap(a, b, min_length=1)
        if olen > 0:
            overlap_graph[(a, b)] = olen
    return overlap_graph
```

Graf ini diimplementasikan sebagai dictionary, di mana kunci adalah pasangan fragmen dan nilai adalah panjang tumpang tindih mereka.

2. Layout

2.1. Fungsi biaya untuk branch and bound

Untuk menentukan langkah terbaik dalam algoritma branch and bound, kita menggunakan fungsi biaya yang menggabungkan panjang superstring saat ini dan panjang tumpang tindih.

```python
def cost_function(length, overlap):
    return length + (1 / overlap if overlap > 0 else float('inf'))
```

2.2. Mengumpulkan superstring menggunakan branch and bound

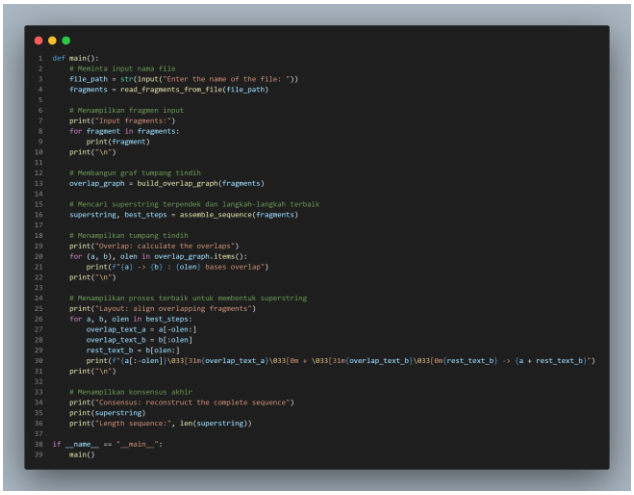
Algoritma branch and bound digunakan untuk menemukan superstring terpendek dengan menggabungkan fragmen-fragmen berdasarkan tumpang tindih mereka. Algoritma ini memotong cabang pencarian yang tidak mungkin menghasilkan superstring yang lebih pendek daripada yang sudah ditemukan:

```python
def assemble_sequence(fragments):
    shortest_superstring = None
    best_steps = []

    # Fungsi rekursif untuk branch and bound
    def branch_and_bound(path, remaining_fragments, steps, current_cost):
        nonlocal shortest_superstring, best_steps
        current_state = "".join(path)

        if not remaining_fragments:
            candidate = "".join(path)
            if shortest_superstring is None or len(candidate) < len(shortest_superstring):
                shortest_superstring = candidate
                best_steps = steps[:]
            return

        # Pruning branch based on the length of the shortest superstring
        if shortest_superstring is not None and len(current_state) >= len(shortest_superstring):
            return

        # Stop further branching if the current cost is infinite
        if current_cost == float('inf'):
            return

        for fragment in remaining_fragments:
            overlap_len = overlap(path[-1], fragment)

            new_path = path[:-1] + [path[-1] + fragment[overlap_len:]]
            new_remaining_fragments = remaining_fragments - {fragment}
            new_steps = steps + [(path[-1], fragment, overlap_len)]
            new_state = "".join(new_path)
            new_cost = cost_function(len(new_state), overlap_len)
            branch_and_bound(new_path, new_remaining_fragments, new_steps, new_cost)

    # Memulai proses rekursif dengan setiap fragmen sebagai titik awal
    for fragment in fragments:
        branch_and_bound([fragment], set(fragments) - {fragment}, [], 0)

    return shortest_superstring, best_steps
```

Inisialisasi algoritma dilakukan dengan shortest_superstring yang menyimpan superstring terpendek yang ditemukan sejauh ini, dan best_steps yang menyimpan langkah-langkah penggabungan yang menghasilkan superstring tersebut. Fungsi rekursif branch_and_bound mencoba menggabungkan fragmen-fragmen dalam berbagai urutan untuk menemukan superstring terpendek. Pada base case, jika tidak ada fragmen yang tersisa, maka kita memeriksa apakah urutan yang dihasilkan lebih pendek dari superstring terpendek yang ditemukan sejauh ini. Proses pruning dilakukan jika panjang superstring saat ini lebih besar atau sama dengan shortest_superstring yang ditemukan, sehingga cabang tersebut dipotong. Algoritma iterasi fragmen mencoba menggabungkan fragmen yang tersisa dan menghitung panjang tumpang tindih. Jika penggabungan berhasil, fungsi ini dipanggil secara rekursif dengan fragmen yang tersisa.

3. Consensus

Fungsi main mengkoordinasikan keseluruhan proses mulai dari membaca fragmen, menghitung tumpang tindih, menyusun urutan fragmen, hingga menghasilkan superstring konsensus:



```python
def main():
    # Meminta input nama file
    file_path = str(input("Enter the name of the file: "))
    fragments = read_fragments_from_file(file_path)

    # Menampilkan fragmen input
    print("Input fragments:")
    for fragment in fragments:
        print(fragment)
    print("\n")

    # Membangun graf tumpang tindih
    overlap_graph = build_overlap_graph(fragments)

    # Mencari superstring terpendek dan langkah-langkah terbaik
    superstring, best_steps = assemble_sequence(fragments)

    # Menampilkan tumpang tindih
    print("Overlap: calculate the overlaps")
    for (a, b), olen in overlap_graph.items():
        print(f"{a} -> {b} : {olen} bases overlap")
    print("\n")

    # Menampilkan proses terbaik untuk membentuk superstring
    print('Layout: align overlapping fragments')
    for a, b, olen in best_steps:
        overlap_text_a = a[-olen:]
        overlap_text_b = b[:olen]
        rest_text_b = b[olen:]
        print(f"{a[:-olen]}\033[31m{overlap_text_a}\033[0m + \033[31m{overlap_text_b}\033[0m{rest_text_b} -> {a + rest_text_b}")
    print("\n")

    # Menampilkan konsensus akhir
    print("Consensus: reconstruct the complete sequence")
    print(superstring)
    print("Length sequence:", len(superstring))

if __name__ == "__main__":
    main()
```

Penjelasan program dimulai dengan langkah membaca fragmen, di mana program mengambil input nama file dan membaca fragmen dari file tersebut. Selanjutnya, pada tahap overlap, program membangun graf tumpang tindih dan menampilkan panjang tumpang tindih antara setiap pasangan fragmen. Setelah itu, pada tahap layout, algoritma branch and bound digunakan untuk menemukan superstring terpendek, dan langkah-langkah penggabungan fragmen ditampilkan. Akhirnya, pada tahap consensus, program menampilkan superstring akhir yang dihasilkan beserta panjangnya.

IV. Uji Coba Kasus

Berikut adalah hasil uji coba kasus dari implementasi program yang sudah dibuat:

Input	Output
AGC CAT TGC ATT	Overlap: calculate the overlaps AGC -> CAT : 1 bases overlap CAT -> TGC : 1 bases overlap CAT -> ATT : 2 bases overlap TGC -> CAT : 1 bases overlap ATT -> TGC : 1 bases overlap Layout: align overlapping fragments AGC + CAT -> AGCAT AGCAT + ATT -> AGCATT AGCATT + TGC -> AGCATTGC Consensus: reconstruct the complete sequence AGCATTGC Length sequence: 8
AGCCA TGAA CAT ATCCT	Overlap: calculate the overlaps AGCCA -> CAT : 2 bases overlap AGCCA -> ATCCT : 1 bases overlap TGAA -> AGCCA : 1 bases overlap TGAA -> ATCCT : 1 bases overlap CAT -> TGAA : 1 bases overlap CAT -> ATCCT : 2 bases overlap ATCCT -> TGAA : 1 bases overlap Layout: align overlapping fragments AGCCA + CAT -> AGCCAT AGCCAT + ATCCT -> AGCCATCCT AGCCATCCT + TGAA -> AGCCATCCTGAA Consensus: reconstruct the complete sequence AGCCATCCTGAA Length sequence: 12
CTCGAA GGTTACCA GTTACCACT TAGGTT CCACTCGAA TACCACT GTTACC	Overlap: calculate the overlaps GGTTACCA -> GTTACCACT : 7 bases overlap GGTTACCA -> CCACTCGAA : 3 bases overlap GGTTACCA -> TACCACT : 5 bases overlap GTTACCACT -> CTCGAA : 2 bases overlap GTTACCACT -> TAGGTT : 1 bases overlap GTTACCACT -> CCACTCGAA : 5 bases overlap GTTACCACT -> TACCACT : 7 bases overlap TAGGTT -> GGTTACCA : 4 bases overlap TAGGTT -> GTTACCACT : 3 bases overlap TAGGTT -> TACCACT : 1 bases overlap TAGGTT -> GTTACC : 3 bases overlap CCACTCGAA -> CTCGAA : 6 bases overlap TACCACT -> CTCGAA : 2 bases overlap TACCACT -> TAGGTT : 1 bases overlap TACCACT -> CCACTCGAA : 5 bases overlap GTTACC -> CTCGAA : 1 bases overlap GTTACC -> GTTACCACT : 6 bases overlap GTTACC -> CCACTCGAA : 2 bases overlap GTTACC -> TACCACT : 4 bases overlap Layout: align overlapping fragments TAGGTT + GTTACC -> TAGGTTACC TAGGTTACC + GGTTACCA -> TAGGTTACCA TAGGTTACCA + GTTACCACT -> TAGGTTACCACT TAGGTTACCACT + TACCACT -> TAGGTTACCACT TAGGTTACCACT + CCACTCGAA -> TAGGTTACCACTCGAA TAGGTTACCACTCGAA + CTCGAA -> TAGGTTACCACTCGAA Consensus: reconstruct the complete sequence TAGGTTACCACTCGAA Length sequence: 16

V. Analisis dan Pembahasan

Berdasarkan hasil uji coba, algoritma mampu menemukan solusi optimal dengan memaksimalkan tumpang tindih antara fragmen, yang merupakan tujuan utama dalam DNA Sequencing Assembly. Algoritma Branch and Bound, meskipun efektif, memiliki tantangan dalam hal efisiensi waktu, terutama ketika berhadapan dengan jumlah fragmen yang sangat besar. Namun, dengan mekanisme pruning yang tepat, algoritma ini mampu memberikan solusi dalam waktu yang relatif singkat untuk ukuran masalah yang sedang. Misalnya, fungsi biaya c(n) = f(n) + 1/g(n) efektif dalam mengevaluasi panjang superstring sementara dan tumpang

tindih, sehingga cabang yang tidak menguntungkan dapat segera dipangkas.

Kompleksitas waktu dari algoritma ini dalam kasus terburuk adalah eksponensial, yaitu O(n!), di mana n adalah jumlah fragmen. Hal ini disebabkan oleh banyaknya kemungkinan urutan fragmen yang harus dieksplorasi. Namun, dengan strategi pruning yang efektif, kompleksitas waktu dapat dikurangi secara signifikan dalam kasus praktis. Dibandingkan dengan metode lain, algoritma ini menawarkan keunggulan dalam hal akurasi solusi, meskipun mungkin memerlukan waktu eksekusi yang lebih lama.

VI. KESIMPULAN

Algoritma Branch and Bound terbukti efektif dalam menghasilkan superstring terpendek dari fragmen-fragmen DNA dengan memaksimalkan tumpang tindih antara fragmen. Hal ini menjadikannya pilihan yang sangat baik untuk aplikasi DNA Sequencing Assembly. Algoritma ini dapat menemukan solusi optimal dengan menggunakan mekanisme pemangkasan yang tepat, yang memungkinkan eliminasi cabang-cabang yang tidak menguntungkan dan dengan demikian mengurangi waktu eksekusi secara signifikan.

VIDEO YOUTUBE

https://youtu.be/14lhRrCsdpo?si=XbCpKhXI-gj1fADL

UCAPAN TERIMA KASIH

Penulis ingin mengucapkan puji syukur kepada Allah SWT karena berkat nikmat dan rahmat-Nya, penulis dapat menyelesaikan makalah ini yang berjudul "Penerapan Algoritma *Branch and Bound* dalam *DNA Sequencing Assembly*" dengan baik. Penulis juga ingin menyampaikan rasa terima kasih yang mendalam kepada kedua orang tua penulis yang telah memberikan dukungan tanpa henti, doa, dan cinta kasih sepanjang proses penulisan makalah ini. Kehadiran mereka menjadi sumber inspirasi dan kekuatan bagi penulis. Selain itu, penulis ingin menyampaikan rasa terima kasih kepada dosen mata kuliah Strategi Algoritma, Dr. Ir. Rinaldi Munir, M.T., Dr. Nur Ulfa Maulidevi, dan Dr. Ir. Rila Mandala, yang telah memberikan bimbingan selama perkuliahan ini. Penulis juga tidak lupa mengucapkan terima kasih kepada teman-teman yang telah memberikan dukungan moral dan semangat selama pembuatan makalah ini.

REFERENSI

[1] GeeksforGeeks. (n.d.). Introduction to Branch and Bound. Diakses pada 10 Juni 2024 dari https://www.geeksforgeeks.org/introduction-to-branch-and-bound-data-structures-and-algorithms-tutorial/.

[2] Munir, R. (2021). Algoritma Branch and Bound 2021 Bagian 1. Diakses pada 10 Juni 2024 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branch-and-Bound-2021-Bagian1.pdf.

[3] Munir, R. (2021). Algoritma Branch and Bound 2021 Bagian 2. Diakses pada 10 Juni 2024 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branchand-Bound-2021-Bagian2.pdf.

[4] Munir, R. (2021). Algoritma Branch and Bound 2021 Bagian 3. Diakses pada 10 Juni 2024 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branchand-Bound-2021-Bagian3.pdf.

[5] Munir, R. (2022). Algoritma Branch and Bound 2022 Bagian 4. Diakses pada 10 Juni 2024 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Branchand-Bound-2022-Bagian4.pdf.

[6] News Medical. (n.d.). DNA Sequence Assembly. Diakses pada 10 Juni 2024 dari https://www.news-medical.net/life-sciences/DNA-Sequence-Assembly.aspx.

[7] Recent Advances in Sequence Assembly: Principles and Applications, Briefings in Functional Genomics, Volume 16, Issue 6, November 2017, Pages 361–378.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Juni 2024

Randy Verdian (13522067)