

Implementasi dan Analisis Perbandingan Efisiensi Algoritma Dijkstra dan Depth-First Search dalam Optimasi Model Lotka-Volterra untuk Dinamika Ekosistem

Muh. Rusmin Nurwadin - 13523068¹

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

rusmn17@gmail.com, 13523068@std.stei.itb.ac.id

Abstrak— Dinamika ekosistem mencerminkan perubahan dan interaksi antara predator dan mangsa, yang berkontribusi pada keseimbangan populasi dalam suatu lingkungan. Model Lotka-Volterra digunakan untuk menggambarkan hubungan ini, namun memerlukan optimasi parameter agar hasil simulasi sesuai dengan kondisi nyata. Kajian ini bertujuan membandingkan efisiensi algoritma Dijkstra dan Depth-First Search (DFS) dalam optimasi parameter model tersebut. Simulasi dilakukan dengan data berbasis model Lotka-Volterra yang dilengkapi noise Gaussian untuk menambahkan elemen realistis. Hasil simulasi menunjukkan bahwa DFS lebih cepat dalam eksplorasi graf, dengan waktu eksekusi 0,004527 detik untuk 5 iterasi pada data berukuran 5000 dengan kompleksitas graf 7. Sebaliknya, Dijkstra menunjukkan performa lebih baik dalam menemukan jalur optimal pada graf berbobot, dengan tingkat error serupa, yaitu sekitar 32,25, pada data berukuran 2000 dengan kompleksitas 5. Pola fluktuasi populasi predator dan mangsa yang khas menggambarkan keseimbangan dinamis ekosistem, di mana populasi mangsa meningkat lebih dahulu sebelum mendukung pertumbuhan predator. Kajian ini memberikan landasan dalam memilih algoritma yang sesuai untuk analisis, baik dari segi efisiensi waktu maupun akurasi hasil.

Kata Kunci— Dinamika ekosistem, Lotka-Volterra, algoritma Dijkstra, Depth-First Search, optimasi parameter

I. PENDAHULUAN

Ekosistem merupakan sistem dinamis yang kompleks, di mana interaksi antara predator dan mangsa menjadi salah satu aspek penting yang sering dipelajari. Salah satu model matematika yang digunakan untuk menggambarkan hubungan ini adalah model Lotka-Volterra, yang mampu memprediksi perubahan populasi dua spesies dalam suatu ekosistem [1]. Model ini, meskipun sederhana, membutuhkan parameter-parameter yang akurat untuk menghasilkan prediksi yang representatif terhadap kondisi nyata.

Proses optimasi parameter dalam model Lotka-Volterra menjadi penting untuk menyesuaikan hasil simulasi dengan data observasi. Dalam konteks ini, algoritma seperti Dijkstra dan Depth-First Search (DFS) digunakan untuk mengoptimalkan parameter berdasarkan data yang dihasilkan. Kedua algoritma tersebut memiliki cara kerja yang berbeda dalam menjelajahi

ruang parameter untuk mencari nilai yang memberikan error minimum terhadap data.

Algoritma Dijkstra bekerja dengan pendekatan jalur terpendek, memanfaatkan struktur graf parameter untuk menemukan solusi optimal berdasarkan error yang dihasilkan [2]. Sementara itu, DFS menggunakan pendekatan eksplorasi menyeluruh untuk menjelajahi semua kemungkinan kombinasi parameter [3]. Perbandingan efisiensi kedua algoritma ini dari segi waktu eksekusi dan hasil optimasi menjadi fokus penelitian.

Penelitian ini bertujuan untuk mengimplementasikan algoritma Dijkstra dan DFS dalam optimasi parameter model Lotka-Volterra, sekaligus menganalisis kinerja kedua algoritma tersebut. Analisis ini dilakukan dengan membandingkan waktu eksekusi dan tingkat error yang dihasilkan masing-masing algoritma, sehingga dapat memberikan gambaran algoritma mana yang lebih efektif digunakan dalam konteks model ekosistem.

Hasil dari penelitian ini diharapkan dapat memberikan wawasan mengenai efektivitas algoritma Dijkstra dan DFS dalam konteks optimasi parameter. Dengan demikian, penelitian ini tidak hanya mendukung pemahaman teoretis, tetapi juga memberikan pendekatan praktis untuk pengelolaan data berbasis model matematika dalam studi ekosistem.

II. LANDASAN TEORI

A. Dinamika Ekosistem

Dinamika ekosistem merujuk pada perubahan dan interaksi yang terjadi di dalam suatu ekosistem sebagai akibat dari faktor biotik (makhluk hidup) dan abiotik (lingkungan fisik). Setiap ekosistem terdiri dari berbagai komponen, seperti produsen, konsumen, dan pengurai, yang saling memengaruhi dalam menjaga keseimbangan [4]. Contohnya, keberadaan predator dan mangsa menciptakan hubungan yang saling bergantung, di mana populasi keduanya dapat berubah secara signifikan tergantung pada ketersediaan sumber daya dan kondisi lingkungan.

Dalam jangka panjang, ekosistem cenderung mencapai titik keseimbangan, di mana populasi organisme relatif stabil. Namun, gangguan seperti bencana alam, perubahan iklim, atau aktivitas manusia, seperti deforestasi dan polusi, dapat

mengubah keseimbangan ini. Misalnya, penurunan populasi predator dapat menyebabkan ledakan populasi mangsa, yang kemudian menguras sumber daya alam secara berlebihan, sehingga merusak struktur ekosistem secara keseluruhan.

Untuk mempelajari dinamika ini, para ilmuwan menggunakan model matematis seperti model Lotka-Volterra, yang menggambarkan interaksi predator-mangsa dalam ekosistem. Model ini membantu memahami pola-pola populasi dan memberikan wawasan tentang bagaimana suatu ekosistem dapat pulih atau berubah setelah terganggu. Melalui pemahaman ini, manusia dapat merancang strategi untuk melindungi keanekaragaman hayati dan menjaga keseimbangan ekosistem.

B. Model Lotka-Volterra

Model Lotka-Volterra merupakan suatu model yang menyajikan dinamika interaksi antara dua populasi yang bersaing untuk persediaan makanan atau sumber alam lainnya, dimana satu populasi berperan sebagai predator, dan yang lainnya sebagai prey dalam suatu ekosistem [5]. Model ini terdiri dari dua persamaan diferensial, yakni sebagai berikut.

1. Dinamika Prey (Mangsa).

$$\frac{dx}{dt} = \alpha x - \beta xy \quad (1)$$

Dengan x = populasi mangsa, y = populasi predator, α = tingkat pertumbuhan mangsa tanpa kehadiran predator, dan β = tingkat kematian mangsa karena interaksi dengan predator.

2. Dinamika Predator.

$$\frac{dy}{dt} = \delta xy - \gamma y \quad (2)$$

Dengan δ = tingkat pertumbuhan predator karena konsumsi mangsa dan γ = tingkat kematian predator tanpa kehadiran mangsa.

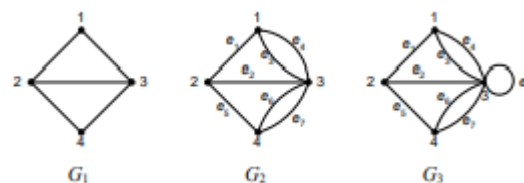
Model ini didasari oleh beberapa asumsi, seperti populasi prey akan tumbuh secara eksponensial tanpa adanya predator, interaksi yang terjadi mengurangi populasi dengan laju proposional terhadap pertemuan kedua populasi, populasi predator akan menurun secara eksponensial tanpa adanya prey, serta predator dapat mengkonversi interaksi dengan prey menjadi pertumbuhan populasinya.

C. Graf

Teori Graf merupakan cabang matematika yang membahas hubungan antar objek, yang direpresentasikan sebagai simpul (node/vertex) dan dihubungkan oleh sisi (edge). Bidang ini menjadi bagian dari matematika diskrit dan bertujuan untuk mempermudah pemahaman hubungan antar objek melalui visualisasi. Graf terdiri dari kumpulan simpul yang bisa saling terhubung atau tidak melalui garis, tanpa memperhatikan ukuran simpul, panjang garis, atau bentuk garisnya, baik lurus maupun melengkung, dan simpul tidak harus berbentuk bulat [6].

Secara matematis, graf didefinisikan sebagai $G = (V, E)$, dengan V adalah himpunan tidak-kosong dari simpul-simpul

(vertices) dan E adalah himpunan sisi yang menghubungkan sepasang simpul [7]. Berikut adalah contoh penggambaran graf.



Gambar 1. Contoh Graf

Sumber : <https://informatika.stei.itb.ac.id/~rinaldi.munir/>

D. Algoritma Dijkstra

Algoritma Dijkstra, ditemukan oleh Edsger Wybe Dijkstra pada tahun 1959, adalah algoritma yang digunakan untuk menemukan jalur terpendek pada graf dengan bobot tidak negatif. Sebagai bagian dari algoritma greedy, Dijkstra sering diterapkan untuk menyelesaikan masalah optimasi. Algoritma ini bekerja dengan mencari bobot paling kecil dalam graf berbobot, sehingga menghasilkan jarak terpendek antara dua atau lebih simpul. Misalnya, pada graf berarah dengan simpul $V(G) = \{v1, v2, \dots, vn\}$ jalur terpendek dari $v1$ ke vn dicari dengan memulai dari $v1$. Algoritma ini secara iteratif memilih simpul dengan jumlah bobot terkecil dari simpul awal, lalu memisahkannya dari perhitungan berikutnya hingga jalur terpendek ditemukan [8].

Secara umum, berikut langkah-langkah dalam menentukan lintasan terpendek pada algoritma dijkstra.

1. Pilih simpul awal sebagai simpul sumber, yang diinisialisasi dengan nilai "1".
2. Buat tabel dengan kolom yang mencakup simpul, status, bobot, dan pendahulu. Isi kolom bobot berdasarkan jarak dari simpul sumber ke semua simpul yang langsung terhubung dengannya.
3. Jika simpul sumber ditemukan, tetapkan sebagai simpul yang dipilih.
4. Berikan label permanen pada simpul terpilih dan perbarui bobot simpul-simpul yang langsung terhubung dengannya.
5. Tentukan simpul sementara berikutnya yang memiliki bobot terkecil untuk proses iterasi selanjutnya.

Dalam model Lotka-Volterra, setiap simpul dapat mewakili kondisi tertentu dari populasi, sedangkan sisi berbobot menggambarkan tingkat perubahan atau hubungan antar kondisi tersebut. Algoritma Dijkstra digunakan untuk mengoptimalkan analisis dinamika ekosistem dengan mencari jalur perubahan populasi paling efisien, yang mencerminkan skenario pertumbuhan atau penurunan populasi dengan dampak minimal terhadap stabilitas ekosistem. Hal ini membantu mengidentifikasi jalur atau interaksi yang paling optimal dalam mendukung keseimbangan ekosistem.

E. Algoritma Depth-First Search (DFS)

Algoritma Depth First Search (DFS) adalah algoritma pencarian yang bergerak secara mendalam dengan memulai dari simpul awal, kemudian melanjutkan kunjungan ke simpul anak paling kiri di tingkat berikutnya [9].

Cara kerja algoritma DFS dimulai dengan memasukkan simpul akar ke dalam sebuah tumpukan. Selanjutnya, ambil simpul teratas dari tumpukan. Jika simpul tersebut adalah solusi, pencarian selesai dan hasil dikembalikan. Jika bukan, masukkan semua simpul yang bertetangga dengan simpul tersebut ke dalam tumpukan. Jika semua simpul telah diperiksa dan tumpukan kosong, pencarian dihentikan dengan hasil bahwa solusi tidak ditemukan. Proses pencarian kemudian diulang dari simpul awal dalam tumpukan.

Algoritma DFS berperan sebagai metode eksplorasi jalur yang dapat digunakan untuk mengeksplorasi semua kemungkinan jalur dalam graf yang merepresentasikan hubungan populasi predator-mangsa dalam model Lotka-Volterra. DFS memungkinkan pencarian jalur alternatif yang lebih mendalam, meskipun tidak secara khusus mencari jalur terpendek seperti Dijkstra. Hal ini relevan dalam situasi di mana analisis perlu mencakup semua kemungkinan perubahan populasi atau interaksi dalam ekosistem.

F. Kompleksitas Algoritma

Kompleksitas algoritma adalah ukuran efisiensi suatu algoritma berdasarkan dua aspek utama, yakni kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu menggambarkan jumlah operasi yang dilakukan oleh algoritma seiring bertambahnya ukuran input, sering dinyatakan menggunakan notasi asimtotik seperti $O(n)$, $O(n^2)$, atau $O(\log n)$. Kompleksitas ruang, di sisi lain, menunjukkan jumlah memori yang diperlukan algoritma untuk menjalankan prosesnya, termasuk variabel sementara, struktur data, dan tumpukan rekursi. Dengan memahami kompleksitas algoritma, kita dapat membandingkan berbagai metode untuk menentukan mana yang lebih optimal untuk digunakan pada masalah tertentu.

Algoritma Dijkstra memiliki kompleksitas waktu $O((V+E)\log V)$ saat menggunakan heap prioritas untuk memilih simpul dengan bobot terkecil. Di sini, V adalah jumlah simpul, dan E adalah jumlah sisi. Proses memasukkan dan menghapus simpul dari heap memerlukan waktu $O(\log V)$, dan setiap sisi diperiksa sekali.

Sebaliknya, algoritma Depth First Search (DFS) memiliki kompleksitas waktu yang lebih sederhana, yaitu $O(V+E)$, karena hanya memerlukan kunjungan linear terhadap semua simpul dan sisi tanpa membutuhkan struktur tambahan seperti heap. Dalam hal kompleksitas ruang, DFS memerlukan $O(V)$ untuk menyimpan visited node dan tumpukan rekursi, sedangkan Dijkstra memerlukan tambahan ruang untuk heap, menjadikan kompleksitas ruangnya $O(V+E)$ tergantung pada implementasi. Dijkstra lebih cocok untuk menemukan jalur terpendek, sedangkan DFS lebih digunakan untuk eksplorasi struktur graf.

III. METODOLOGI

Penelitian ini diawali dengan studi literatur mengenai dinamika ekosistem, model Lotka-Volterra, serta algoritma Dijkstra dan Depth-First Search (DFS). Studi ini bertujuan untuk memahami teori dasar yang mendukung optimasi parameter dalam model Lotka-Volterra, termasuk bagaimana graf dapat digunakan untuk memetakan kombinasi parameter dan menjelajahi solusinya menggunakan kedua algoritma

tersebut. Selain itu, dilakukan analisis literatur mengenai kompleksitas algoritma untuk memperkuat kerangka evaluasi efisiensi.

Data yang digunakan dalam penelitian ini adalah data fiktif yang digenerasikan sendiri. Data ini mencerminkan populasi predator dan mangsa yang disimulasikan menggunakan persamaan model Lotka-Volterra dengan tambahan noise Gaussian untuk merepresentasikan fluktuasi alami dalam ekosistem. Data ini dirancang untuk memastikan eksperimen dapat dilakukan secara sistematis dan terkontrol. Graf parameter yang dibangun dari kombinasi nilai parameter model yang digunakan untuk mengeksplorasi parameter terbaik melalui algoritma Dijkstra dan DFS.

Dalam penelitian ini, beberapa aspek divariasikan untuk evaluasi performa algoritma, yaitu jumlah titik waktu (n) dalam data, jumlah node dan edge dalam graf parameter, serta kombinasi parameter model. Variasi ini dirancang untuk mengukur pengaruh ukuran data dan kompleksitas graf terhadap efisiensi waktu eksekusi dan akurasi hasil optimasi. Evaluasi dilakukan dengan membandingkan hasil kedua algoritma dalam menemukan parameter optimal berdasarkan error terkecil dan kecepatan eksekusi pada berbagai skenario.

IV. IMPLEMENTASI

A. Library



```
1 import numpy as np
2 from scipy.integrate import odeint
3 import networkx as nx
4 import heapq
5 import matplotlib.pyplot as plt
6 import time
7
```

Gambar 2. Library
Sumber : Dokumen Pribadi

Implementasi menggunakan beberapa library yang sangat penting untuk mendukung simulasi dan analisis data. *NumPy* digunakan untuk memproses array dan melakukan operasi numerik, seperti membuat noise Gaussian untuk menambahkan variasi pada data simulasi. *SciPy* menyediakan fungsi *odeint*, yang membantu menyelesaikan persamaan diferensial non-linear dalam model Lotka-Volterra. *NetworkX* berperan penting dalam membangun graf parameter untuk analisis kombinasi parameter, sementara *Heapq* digunakan dalam algoritma Dijkstra untuk mengelola prioritas node dengan efisiensi tinggi. *Matplotlib* dipakai untuk membuat visualisasi data hasil simulasi, dan *Time* digunakan untuk mencatat waktu eksekusi setiap algoritma, yang berguna untuk membandingkan efisiensi antara algoritma Dijkstra dan DFS.

B. Model Lotka-Volterra

```
1 def lotka_volterra(state, t, alpha, beta, delta, gamma):
2     prey, predator = state
3     dprey_dt = alpha * prey - beta * prey * predator
4     dpredator_dt = delta * prey * predator - gamma * predator
5     return [dprey_dt, dpredator_dt]
6
```

Gambar 3. Model Lotka-Volterra
Sumber : Dokumen Pribadi

Bagian ini diimplementasikan dalam fungsi `lotka_volterra()`. Fungsi ini mendeskripsikan interaksi antara populasi mangsa dan predator dalam bentuk persamaan diferensial. Persamaan ini menghitung laju perubahan populasi mangsa yang dipengaruhi oleh tingkat pertumbuhan alami dan interaksi dengan predator. Sementara itu, laju perubahan populasi predator dihitung berdasarkan konsumsi mangsa dan tingkat kematian alami predator. Fungsi ini menjadi inti dari simulasi, karena semua data populasi dihasilkan berdasarkan model ini.

C. Calculate Error

```
1 def calculate_error(params, data, t):
2     alpha, beta, delta, gamma = params
3     initial_state = [data[0], 0], data[0], 1]
4     solution = odeint(lotka_volterra, initial_state, t, args=(alpha, beta, delta, gamma))
5     return np.mean((solution - data)**2)
6
```

Gambar 4. Calculate Error
Sumber : Dokumen Pribadi

Fungsi `calculate_error()` dirancang untuk mengukur seberapa akurat kombinasi parameter tertentu dalam mencocokkan data sintetik yang dihasilkan. Parameter yang diberikan digunakan untuk menjalankan simulasi model Lotka-Volterra, menghasilkan data populasi mangsa dan predator. Setelah itu, hasil simulasi dibandingkan dengan data sintetik menggunakan rata-rata kuadrat selisih (*mean squared error*). Error ini menjadi indikator apakah parameter yang diuji mendekati nilai sebenarnya atau tidak.

D. Generate Data

```
1 def generate_data(n_points, noise_level=0.1):
2     true_params = {
3         'alpha': 0.8,
4         'beta': 0.08,
5         'delta': 0.07,
6         'gamma': 0.45
7     }
8
9     t = np.linspace(0, 100, n_points)
10    initial_state = [10.0, 5.0]
11
12    solution = odeint(lotka_volterra, initial_state, t,
13                    args=(true_params['alpha'], true_params['beta'],
14                        true_params['delta'], true_params['gamma']))
15
16    noise = np.random.normal(0, noise_level, solution.shape)
17    noisy_data = solution + noise
18    noisy_data = np.maximum(noisy_data, 0)
19
20    return t, noisy_data, true_params
21
```

Gambar 5. Generate Data
Sumber : Dokumen Pribadi

Fungsi `generate_data()` digunakan untuk membuat data sintetik populasi mangsa dan predator. Pertama, fungsi ini

memanfaatkan model Lotka-Volterra untuk menghasilkan data simulasi tanpa noise. Setelah itu, ditambahkan *noise Gaussian* untuk mensimulasikan fluktuasi alami yang biasa terjadi dalam data nyata, seperti perubahan ekosistem yang tidak terduga. Fungsi ini juga memungkinkan variasi jumlah data (n_points), yang mempermudah pengujian algoritma pada dataset yang lebih kecil atau lebih besar. Hasil akhirnya berupa waktu (t), data populasi dengan noise, dan parameter awal yang digunakan dalam simulasi.

E. Graf Parameter

```
1 def create_parameter_graph(data, t, complexity=5):
2     G = nx.Graph()
3     param_ranges = {
4         'alpha': np.linspace(0.5, 1.5, complexity),
5         'beta': np.linspace(0.05, 0.15, complexity),
6         'delta': np.linspace(0.05, 0.1, complexity),
7         'gamma': np.linspace(0.3, 0.7, complexity)
8     }
9
10    param_combinations = []
11    for alpha in param_ranges['alpha']:
12        for beta in param_ranges['beta']:
13            for delta in param_ranges['delta']:
14                for gamma in param_ranges['gamma']:
15                    params = (alpha, beta, delta, gamma)
16                    param_combinations.append(params)
17
18    for i, params in enumerate(param_combinations):
19        error = calculate_error(params, data, t)
20        G.add_node(i, params=params, error=error)
21
22    for i in range(len(param_combinations)):
23        for j in range(i+1, len(param_combinations)):
24            diff = np.sum(np.abs(np.array(param_combinations[i]) - np.array(param_combinations[j])))
25            if diff < 0.1:
26                weight = abs(G.nodes[i]['error'] - G.nodes[j]['error'])
27                G.add_edge(i, j, weight=weight)
28
29    return G, param_combinations
```

Gambar 6. Graf Parameter
Sumber : Dokumen Pribadi

Fungsi `create_parameter_graph()` digunakan untuk membangun graf parameter yang memetakan kombinasi nilai parameter model. Setiap node dalam graf merepresentasikan satu kombinasi parameter, sedangkan edge menghubungkan node berdasarkan kedekatan nilai parameter. Bobot edge dihitung berdasarkan selisih error antara dua node yang terhubung. Fungsi ini memungkinkan pengaturan kompleksitas graf melalui *parameter complexity*, yang menentukan jumlah kombinasi parameter yang dihasilkan. Graf ini menjadi dasar bagi algoritma Dijkstra dan DFS untuk mengeksplorasi kombinasi parameter terbaik.

F. Optimasi Dijkstra

```
1 def dijkstra_optimization(G, param_combinations):
2     start_time = time.perf_counter()
3
4     start_node = 0
5     distances = {node: float('inf') for node in G.nodes()}
6     distances[start_node] = G.nodes[start_node]['error']
7     pq = [(G.nodes[start_node]['error'], start_node)]
8     best_node = start_node
9
10    while pq:
11        current_distance, current_node = heapq.heappop(pq)
12
13        if current_distance > distances[current_node]:
14            continue
15
16        if G.nodes[current_node]['error'] < G.nodes[best_node]['error']:
17            best_node = current_node
18
19        for neighbor in G.neighbors(current_node):
20            distance = current_distance + G[current_node][neighbor]['weight']
21
22            if distance < distances[neighbor]:
23                distances[neighbor] = distance
24                heapq.heappush(pq, (distance, neighbor))
25
26    end_time = time.perf_counter()
27    return param_combinations[best_node], G.nodes[best_node]['error'], end_time - start_time
28
```

Gambar 7. Optimasi Dijkstra
Sumber : Dokumen Pribadi

Fungsi `dijkstra_optimization()` mengimplementasikan algoritma Dijkstra untuk mencari kombinasi parameter dengan

nilai error terkecil. Algoritma ini memulai eksplorasi dari node awal dan menggunakan heap untuk memilih jalur dengan bobot terkecil secara efisien. Pada setiap langkah, algoritma membandingkan error dari node yang sedang dieksplorasi dengan node terbaik yang ditemukan sejauh ini. Fungsi ini mengembalikan kombinasi parameter terbaik, nilai error terkecil, dan waktu eksekusi, yang nantinya akan dianalisis untuk membandingkan performa.

G. Optimasi DFS

```

1 def dfs_optimization(G, param_combinations):
2     start_time = time.perf_counter()
3
4     visited = set()
5     best_params = None
6     best_error = float('infinity')
7
8     def dfs_recursive(node):
9         nonlocal best_params, best_error
10        visited.add(node)
11
12        current_error = G.nodes[node]['error']
13        if current_error < best_error:
14            best_error = current_error
15            best_params = param_combinations[node]
16
17        for neighbor in G.neighbors(node):
18            if neighbor not in visited:
19                dfs_recursive(neighbor)
20
21    dfs_recursive(0)
22    end_time = time.perf_counter()
23    return best_params, best_error, end_time - start_time
24

```

Gambar 8. Optimasi DFS
Sumber : Dokumen Pribadi

Fungsi *dfs_optimization()* menggunakan algoritma Depth-First Search (DFS) untuk mengeksplorasi graf parameter. Berbeda dengan Dijkstra, DFS menjelajahi graf secara mendalam tanpa memprioritaskan jalur dengan bobot terkecil. Algoritma ini mencatat setiap error yang ditemukan dan menyimpannya jika error tersebut lebih kecil dari error terbaik sebelumnya. Meskipun tidak seefisien Dijkstra dalam graf berbobot, DFS tetap bisa menemukan parameter terbaik dengan eksplorasi menyeluruh. Hasilnya adalah parameter terbaik, nilai error terkecil, dan waktu eksekusi.

H. Program Utama

```

1 def main():
2     # Ukuran data yang akan diuji
3     data_sizes = [100, 500, 1000, 2000, 5000]
4     # Kompleksitas graf (jumlah sisi per parameter)
5     graph_complexities = [3, 4, 5, 6, 7]
6
7     print("\nAnalisis Performa dengan Berbagai Ukuran Data dan Kompleksitas Graf:")
8     print("\nUkuran Data | Kompleksitas | Nodes | Edges | Waktu Dijkstra | Waktu DFS | Error Dijkstra | Error DFS")
9     print("-" * 100)
10
11    for n in data_sizes:
12        for complexity in graph_complexities:
13            # Generate data
14            t, data, true_params = generate_data(n)
15
16            # Buat graf
17            G, param_combinations = create_parameter_graph(data, t, complexity)
18
19            # Optiasi dengan Dijkstra
20            best_params_dijkstra, error_dijkstra, time_dijkstra = dijkstra_optimization(G, param_combinations)
21
22            # Optiasi dengan DFS
23            best_params_dfs, error_dfs, time_dfs = dfs_optimization(G, param_combinations)
24
25            print(f"{n:10d} | {complexity:10d} | {G.number_of_nodes():10d} | {G.number_of_edges():10d} | "
26                  f"time_dijkstra:~14.6f | {time_dijkstra:~9.6f} | {error_dijkstra:~14.6f} | {error_dfs:~9.6f}")
27
28    # Demonstrasi multiple runs untuk ukuran dan kompleksitas terbesar
29    if n == data_sizes[-1] and complexity == graph_complexities[-1]:
30        print(f"\nDemonstrasi Multiple Runs untuk n = {n}, kompleksitas = {complexity}:")
31        num_runs = 5
32
33        # Multiple runs untuk Dijkstra
34        start_time = time.perf_counter()
35        for _ in range(num_runs):
36            _ = dijkstra_optimization(G, param_combinations)
37        dijkstra_multi_time = time.perf_counter() - start_time
38
39        # Multiple runs untuk DFS
40        start_time = time.perf_counter()
41        for _ in range(num_runs):
42            _ = dfs_optimization(G, param_combinations)
43        dfs_multi_time = time.perf_counter() - start_time
44
45        print(f"\nWaktu untuk {num_runs} runs:")
46        print(f"Dijkstra: {dijkstra_multi_time:~6f} s")
47        print(f"DFS : {dfs_multi_time:~6f} s")
48
49    # Plot contoh hasil untuk ukuran data menengah
50    mid_size = data_sizes[len(data_sizes)//2]
51    t, data, _ = generate_data(mid_size)
52
53    plt.figure(figsize=(12, 6))
54    plt.plot(t, data[:, 0], 'b.', label='Data Mangsa', alpha=0.5)
55    plt.plot(t, data[:, 1], 'r.', label='Data Predator', alpha=0.5)
56    plt.title(f'Data Example (n={mid_size})')
57    plt.xlabel('Waktu')
58    plt.ylabel('Populasi')
59    plt.legend()
60    plt.grid(True)
61    plt.show()
62

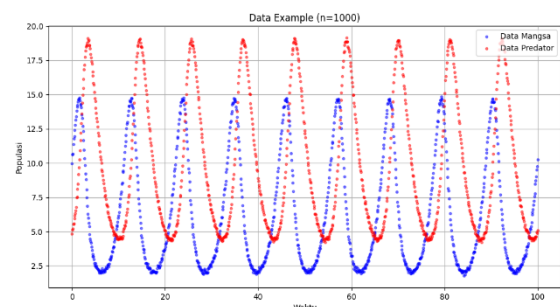
```

Gambar 9. Program Utama
Sumber : Dokumen Pribadi

Fungsi *main()* menyatukan semua komponen dalam kode untuk menjalankan pengujian algoritma. Variasi ukuran data (*data_sizes*) dan kompleksitas graf (*graph_complexities*) diuji untuk melihat bagaimana algoritma Dijkstra dan DFS berperfaoma dalam berbagai kondisi. Hasilnya dicetak dalam bentuk tabel yang mencakup waktu eksekusi, error terkecil, jumlah node, dan jumlah edge pada graf. Selain itu, untuk ukuran data dan kompleksitas graf terbesar, dilakukan pengujian *multiple runs* untuk mengevaluasi stabilitas waktu eksekusi masing-masing algoritma. Sebagai tambahan, hasil simulasi divisualisasikan menggunakan Matplotlib untuk memberikan gambaran tentang dinamika populasi mangsa dan predator yang dihasilkan oleh model Lotka-Volterra.

V. HASIL DAN PEMBAHASAN

Berdasarkan implementasi program yang telah dibuat, diperoleh visualisasi data sebagai berikut.



Gambar 10. Data Fiktif untuk n = 1000
Sumber: Dokumen Pribadi

Seperti yang ditunjukkan pada Gambar 10, grafik memperlihatkan pola fluktuasi populasi mangsa dan predator berdasarkan simulasi model Lotka-Volterra dengan jumlah data $n = 1000$. Pola ini mencerminkan hubungan saling ketergantungan antara kedua populasi yang sangat khas dalam ekosistem predator-mangsa. Populasi mangsa yang meningkat secara langsung memberikan kesempatan bagi populasi predator untuk tumbuh, meskipun peningkatan predator tidak terjadi secara instan melainkan setelah jeda waktu tertentu. Fenomena ini masuk akal karena predator memerlukan waktu untuk bereproduksi dan memanfaatkan ketersediaan mangsa sebagai sumber makanan.

Namun, setelah populasi predator mencapai puncaknya, tekanan predasi terhadap mangsa menjadi sangat besar sehingga menyebabkan populasi mangsa menurun drastis. Penurunan ini, pada gilirannya, memengaruhi populasi predator yang lambat laun juga menurun akibat berkurangnya ketersediaan sumber makanan. Proses ini terus berulang, menciptakan pola fluktuasi periodik yang terlihat pada grafik. Tambahan noise Gaussian pada simulasi ini, seperti yang tampak pada variasi kecil di puncak dan lembah grafik di Gambar 10, menambah elemen realistis yang menggambarkan kondisi dunia nyata, seperti variasi cuaca, perubahan lingkungan, atau faktor genetik dalam populasi.

Salah satu aspek menarik dari grafik ini adalah keseimbangan dinamis yang tercipta. Kedua populasi tidak pernah mencapai nol, menunjukkan bahwa meskipun terjadi tekanan predasi atau kekurangan makanan, selalu ada cukup individu yang bertahan untuk memulai siklus berikutnya. Selain itu, hubungan fase antara kedua populasi juga sangat jelas terlihat. Populasi mangsa selalu mencapai puncaknya sebelum predator, sedangkan populasi predator mencapai titik terendahnya setelah populasi mangsa berada di angka minimum. Pola seperti ini memberikan gambaran yang mendalam tentang bagaimana interaksi antara predator dan mangsa menjaga kestabilan ekosistem dalam jangka panjang.

Analisis Performa dengan Berbagai Ukuran Data dan Kompleksitas Graf:

Ukuran Data	Kompleksitas	Nodes	Edges	Waktu Dijkstra	Waktu DFS	Error Dijkstra	Error DFS
100	3	81	252	0.000081	0.000014	37.801758	37.801758
100	4	256	1600	0.000185	0.000021	34.293607	34.293607
100	5	625	6775	0.000719	0.000060	32.558918	32.558918
100	6	1296	24984	0.004766	0.000393	1.014923	1.014923
100	7	2461	79842	0.014027	0.001018	1.587725	1.587725
500	3	81	252	0.000073	0.000010	38.100489	38.100489
500	4	256	1600	0.000187	0.000018	34.554299	34.554299
500	5	625	6775	0.000738	0.000059	32.623557	32.623557
500	6	1296	24984	0.005121	0.000332	1.004026	1.004026
500	7	2461	79842	0.013584	0.000986	1.125063	1.125063
1000	3	81	252	0.000067	0.000009	38.321566	38.321566
1000	4	256	1600	0.000181	0.000017	34.915623	34.915623
1000	5	625	6775	0.000733	0.000060	32.771488	32.771488
1000	6	1296	24984	0.004708	0.000330	1.020454	1.020454
1000	7	2461	79842	0.013098	0.000925	1.001893	1.001893
2000	3	81	252	0.000070	0.000009	38.123886	38.123886
2000	4	256	1600	0.000188	0.000017	34.533741	34.533741
2000	5	625	6775	0.000707	0.000061	31.703667	31.703667
2000	6	1296	24984	0.010855	0.000616	1.058730	1.058730
2000	7	2461	79842	0.013629	0.000997	1.702381	1.702381
5000	3	81	252	0.000077	0.000009	38.204539	38.204539
5000	4	256	1600	0.000194	0.000018	34.507674	34.507674
5000	5	625	6775	0.000735	0.000060	32.250726	32.250726
5000	6	1296	24984	0.004817	0.000381	1.417858	1.417858
5000	7	2461	79842	0.015303	0.001097	1.031042	1.031042

Gambar 11. Analisis Performa
Sumber: Dokumen Pribadi

Selanjutnya adalah performa algoritma, analisis yang disajikan dalam Gambar 11 memberikan informasi penting tentang bagaimana ukuran data dan kompleksitas graf memengaruhi efisiensi algoritma Dijkstra dan DFS. Ketika ukuran data dan kompleksitas graf meningkat, jumlah node dan edge dalam graf bertambah secara eksponensial. Sebagai contoh, pada data dengan ukuran 100 dan kompleksitas 3, graf memiliki 81 node dan 252 edge. Namun, ketika kompleksitas

naik menjadi 7, jumlah node melonjak hingga 2401 dengan 79842 edge. Peningkatan signifikan ini menunjukkan bagaimana ukuran data yang besar dapat memengaruhi beban komputasi pada kedua algoritma.

Waktu eksekusi juga sangat dipengaruhi oleh kompleksitas graf. Seperti yang terlihat pada Gambar 11, DFS menunjukkan efisiensi waktu yang jauh lebih baik dibandingkan Dijkstra, terutama pada graf sederhana. Contohnya, untuk data berukuran 5000 dengan kompleksitas 7, DFS menyelesaikan proses hanya dalam waktu 0,001 detik, sedangkan Dijkstra membutuhkan 0,015 detik. Hal ini disebabkan oleh sifat DFS yang hanya fokus pada eksplorasi graf tanpa perlu memperhitungkan bobot jalur, menjadikannya sangat cepat untuk tugas eksplorasi. Sebaliknya, Dijkstra, yang dirancang untuk menghitung jalur terpendek dengan memperhitungkan bobot, memerlukan waktu lebih lama, terutama pada graf yang lebih kompleks.

```
Demonstrasi Multiple Runs untuk n = 5000, kompleksitas = 7:
Waktu untuk 5 runs:
Dijkstra: 0.068689 s
DFS      : 0.004527 s
```

Gambar 12. Demonstrasi Performa
Sumber: Dokumen Pribadi

Efisiensi waktu algoritma ini semakin jelas ketika melihat hasil demonstrasi pada Gambar 12, yang menunjukkan total waktu eksekusi untuk 5 kali iterasi pada data berukuran 5000 dengan kompleksitas 7. DFS membutuhkan waktu hanya 0,004527 detik untuk menyelesaikan semua iterasi, sedangkan Dijkstra membutuhkan 0,068689 detik. Hasil ini menggarisbawahi bahwa DFS sangat ideal untuk eksplorasi graf yang membutuhkan respons cepat, sementara Dijkstra lebih cocok untuk kebutuhan analisis mendalam yang mengutamakan akurasi jalur terpendek.

Dari segi tingkat kesalahan atau error, kedua algoritma menunjukkan hasil yang serupa dalam banyak skenario. Sebagai contoh, pada data berukuran 2000 dengan kompleksitas 5, error relatif untuk Dijkstra dan DFS hampir sama, yaitu sekitar 32,25. Temuan ini menunjukkan bahwa meskipun DFS lebih cepat, ia tetap mampu menghasilkan hasil yang cukup akurat untuk banyak aplikasi. Namun, Dijkstra tetap menjadi pilihan utama untuk graf yang memerlukan hasil presisi tinggi, terutama jika bobot jalur menjadi faktor yang signifikan.

Secara keseluruhan, gambar dan tabel yang disajikan memberikan gambaran yang mendalam tentang karakteristik kedua algoritma. DFS menawarkan efisiensi luar biasa dalam hal waktu eksekusi, menjadikannya pilihan yang sangat baik untuk eksplorasi cepat pada graf sederhana. Di sisi lain, Dijkstra memberikan keunggulan dalam akurasi jalur terpendek, meskipun membutuhkan waktu komputasi yang lebih besar. Oleh karena itu, pemilihan algoritma sebaiknya disesuaikan dengan kebutuhan spesifik, baik itu efisiensi waktu maupun tingkat presisi hasil.

VI. KESIMPULAN

Algoritma Dijkstra dan Depth-First Search (DFS) menunjukkan performa yang berbeda dalam optimasi parameter model Lotka-Volterra. Pada data berukuran 5000 dengan kompleksitas graf 7, DFS menyelesaikan proses dalam waktu 0,004527 detik untuk 5 iterasi, sementara Dijkstra membutuhkan 0,068689 detik. DFS lebih unggul dalam hal

efisiensi waktu, terutama pada graf sederhana, tetapi Dijkstra lebih akurat dalam menemukan jalur optimal, khususnya pada graf berbobot. Pada data berukuran 2000 dengan kompleksitas 5, kedua algoritma menghasilkan error relatif yang hampir sama, yaitu sekitar 32,25, menunjukkan bahwa DFS tetap dapat memberikan hasil yang cukup baik meski lebih cepat.

Simulasi menunjukkan pola fluktuasi populasi predator dan mangsa yang khas. Populasi mangsa meningkat lebih dulu, mendukung pertumbuhan populasi predator, sebelum akhirnya menurun akibat tekanan predasi. Pola ini berulang secara periodik, mencerminkan keseimbangan dinamis dalam ekosistem. Penambahan noise Gaussian pada simulasi menambah elemen realistis yang menggambarkan fluktuasi alami seperti variasi lingkungan atau genetika.

Hasil penelitian ini menunjukkan bahwa Dijkstra lebih cocok digunakan untuk kebutuhan analisis yang membutuhkan tingkat akurasi tinggi, sementara DFS lebih efektif untuk eksplorasi cepat. Temuan ini memberikan pemahaman yang lebih mendalam mengenai efektivitas kedua algoritma dalam optimasi parameter model Lotka-Volterra, sekaligus mendukung upaya pengelolaan ekosistem berbasis pendekatan matematika secara lebih terarah dan efisien.

VII. LAMPIRAN

Github : <https://github.com/Rusmn/Makalah-Matematika-Diskrit>

VIII. SARAN DAN UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih atas berkah yang senantiasa diberikan oleh Allah SWT, karena-Nya makalah ini dapat terselesaikan tanpa ada hambatan yang berarti. Selanjutnya, penulis juga mengucapkan terima kasih kepada kedua orang tua penulis yang selalu mendukung dan mendoakan selama proses pengerjaan makalah ini. Penulis juga menyampaikan terima kasih kepada Ir. Rila Mandala, M.Sc., Ph.D., selaku dosen pengampu mata kuliah IF1220 Matematika Diskrit, yang telah memberikan arahan dan materi-materi yang mendasari penelitian ini. Terakhir, penghargaan juga diberikan kepada rekan-rekan penulis atas masukan dan diskusi yang sangat membantu selama pengerjaan makalah ini.

Untuk pengembangan penelitian lebih lanjut, penulis menyarankan beberapa hal berikut.

1. Penggunaan dataset nyata dari ekosistem tertentu untuk menguji validitas model Lotka-Volterra secara lebih komprehensif.
2. Analisis lebih mendalam terhadap algoritma lain, seperti algoritma genetika atau simulated annealing, sebagai pembanding dengan algoritma Dijkstra dan DFS.
3. Pengembangan implementasi yang mendukung pemrosesan paralel untuk meningkatkan efisiensi pada data berskala besar.

Diharapkan dengan saran-saran ini, penelitian selanjutnya dapat memberikan analisis yang lebih komprehensif serta implementasi yang lebih efektif untuk optimasi model matematika dalam dinamika ekosistem.


REFERENCES

- [1] S. I. Salwa, L. A. Shakira and D. Savitri, "DINAMIKA MODEL MANGSA-PEMANGSA LOTKA VOLTERRA DENGAN ADANYA KERJA SAMA BERBURU PADA PEMANGSA," *Jurnal Riset dan Aplikasi Matematika (JRAM)*, vol. 7, no. 2, pp. 195-205, 2023.
- [2] F. Kuncoro, I. A. Zulkarnain and G. A. Buntoro, "Penerapan Algoritma Dijkstra Dalam Menentukan Rute Terpendek pada TPA Mrican," *ANTIVIRUS: Jurnal Ilmiah Teknik Informatika*, vol. 18, no. 2, pp. 200-211, 2024.
- [3] D. Tamara, "DFS (Depth First Search) : Pengertian, Kekurangan, Kelebihan, dan Contohnya," Medium.com, 16 10 2021. [Online]. Available: <https://medium.com/@defytamara2610/dfs-depth-first-search-pengertian-kekurangan-kelebihan-dan-contohnya-2b9b1eee2b3d>. [Accessed 7 1 2025].
- [4] Sampoerna Academy, "Pengertian Dinamika Populasi, Penyebab Ekosistem dan Habitat," Sampoerna Academy, 2024. [Online]. Available: <https://www.sampoernaacademy.sch.id/news/dinamika-populasi>. [Accessed 7 January 2025].
- [5] R. Monica, L. Deswita and R. Pane, "KESTABILAN POPULASI MODEL LOTKA-VOLTERRA TIGA SPESIES DENGAN TITIK KESETIMBANGAN," *JOM FMIPA*, vol. 1, no. 2, pp. 133-141, 2014.
- [6] Humas Tel-U Surabaya, "Teori Graf: Sejarah, Manfaat, dan Aplikasinya," Telkom University, 30 Oktober 2023. [Online]. Available: <https://surabaya.telkomuniversity.ac.id/teori-graf-sejarah-manfaat-dan-aplikasinya/>. [Accessed 7 Januari 2025].
- [7] R. Munir, "Graf (Bag. 1)," 2024. [Online]. Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>. [Accessed 7 Januari 2025].
- [8] M. C. Bunaen, H. Pratiwi and Y. Riti, "PENERAPAN ALGORITMA DIJKSTRA UNTUK MENENTUKAN RUTE TERPENDEK DARI PUSAT KOTA SURABAYA KE TEMPAT BERSEJARAH," *Jurnal Teknologi Dan Sistem Informasi Bisnis*, vol. 4, no. 1, pp. 213-223, 2022.
- [9] D. Tamara, "DFS (Depth First Search) : Pengertian, Kekurangan, Kelebihan, dan Contohnya," 16 Oktober 2021. [Online]. Available: <https://medium.com/@defytamara2610/dfs-depth-first-search-pengertian-kekurangan-kelebihan-dan-contohnya-2b9b1eee2b3d>. [Accessed 7 Januari 2025].

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, 8 Januari 2025



Muh. Rusmin Nurwadin, 13523068