LAPORAN TUGAS BESAR STRUKTUR DATA

IMPLEMENTASI ALGORITMA DIJKSTRA UNTUK OPTIMASI RUTE PENGIRIMAN PADA SISTEM LOGISTIK

Disusun Oleh:

Satria Putra Dharma Prayudha - 21104036

Yogi Hafidh Maulana - 2211104061

Ganesha Rahman Gibran - 2211104058

SE07-02

Dosen :

Wahyu Andi Saputra, S.Pd., M.Eng

PROGRAM STUDI S1 REKAYASA PERANGKAT LUNAK

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2024

A. Tugas Yang Akan Dilakukan :

1) Membuat Representasi Graf

a) Membangun struktur graf yang terdiri dari simpul (kota) dan rute yang dilalui.

b) Menentukan bobot pada setiap rute untuk mewakili biaya pengiriman antar kota.

2) Implementasi Salah Satu Algoritma

a) Mengimplementasikan salah satu algoritma pada graf, digunakan algoritma dijkstra dalam pembuatan graf. Untuk detail fungsi :

- Dijkstra: Fungsi Dijkstra digunakan untuk mencari rute terpendek antara dua titik (gudang) dalam jaringan distribusi. Dengan menggunakan algoritma Dijkstra, sistem dapat menghitung jarak minimal yang harus ditempuh untuk mengirimkan barang dari satu gudang ke gudang lainnya, memastikan efisiensi dalam pengiriman.

3) Membuat fungsi lain yang berkaitan dengan problem yang terkait dengan studi kasus yang dipilih yaitu :

- **Studi Kasus 3: Rute Optimal untuk Pengiriman Barang**

"Sebuah perusahaan logistik memiliki jaringan pengiriman antara gudang-gudang di berbagai kota. Setiap gudang adalah node, dan rute antar gudang adalah edge dengan bobot yang mewakili biaya atau jarak tempuh. Tugas Anda adalah merancang sistem untuk meminimalkan biaya pengiriman barang antar kota."

Detail Fungsi Lainnya :

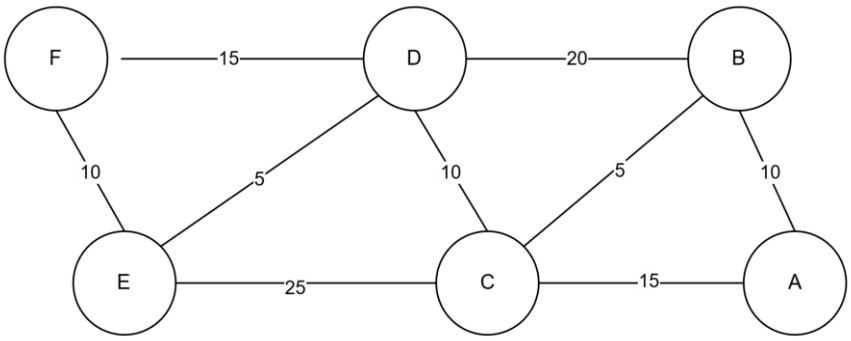
- **Jalur Alternatif dengan Node Diblokir**: Fungsi ini berguna untuk mencari jalur alternatif ketika salah satu jalur atau node (gudang) tidak dapat digunakan, misalnya karena gangguan atau kerusakan. Sistem akan menghitung rute baru yang menghindari node yang diblokir, menjamin pengiriman tetap berlangsung meskipun ada hambatan.

- **TSP (Traveling Salesman Problem)**: Fungsi ini digunakan untuk mencari atau menentukkan rute paling efisien yang dimana rute ini mengunjungi semua gudang (node) dalam satu perjalanan, dimulai dan berakhir di gudang yang sama. Ini penting untuk mengoptimalkan rute pengiriman yang mengunjungi banyak lokasi, mengurangi biaya dan waktu perjalanan dalam distribusi barang.

B. Luaran Tugas

Dari rancangan tugas yang sudah ada, maka didapatkan luaran sebagai berikut :

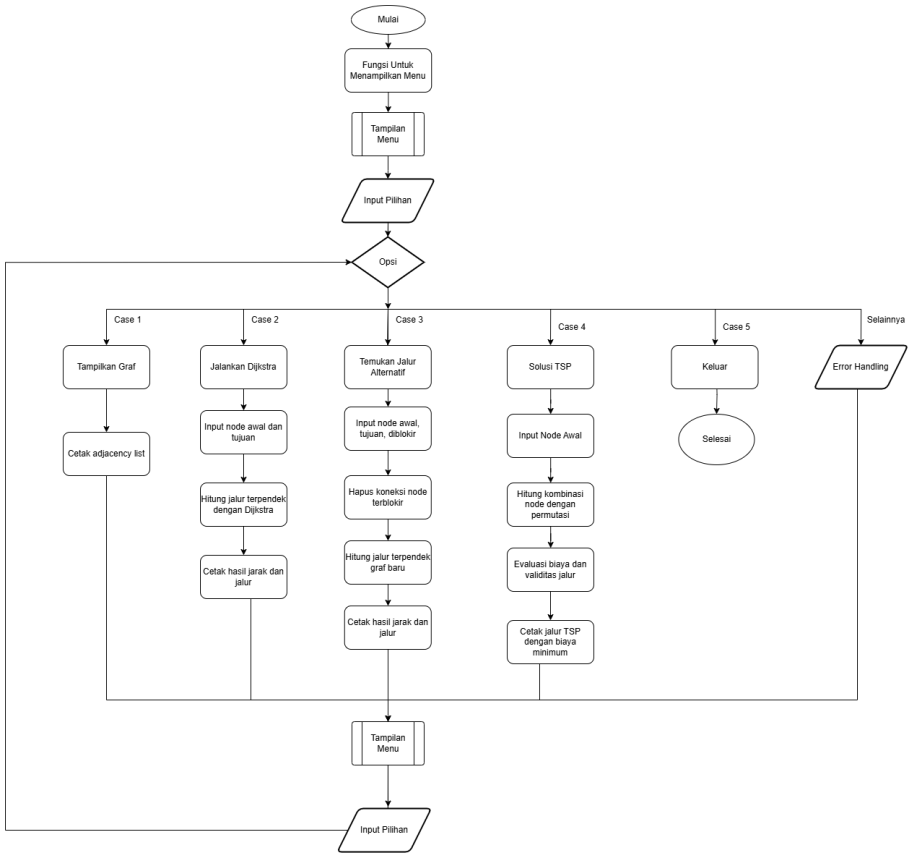
a. Ilustrasi Graf



Ilustrasi Graf tersebut menggambarkan jaringan Gudang (Node) yang dihubungkan dengan tepi berarah tak berarah yang menggambarkan hubungan antar gudang di antar kota (Node), dimana setiap node dilambangkan dengan huruf, seperti A, B, C, D, E, dan F.

Dari garis - garis yang menghubungkan antar node tersebut menunjukkan adanya jalur dengan bobot tertentu (dalam bentuk jarak atau biaya), yang tercantum di sepanjang garis. Misalnya, jarak antara node A dan B adalah 10, sedangkan antara C dan E adalah 25. Dari struktur graf ini dapat digunakan untuk analisis rute terpendek, pencarian jalur alternatif, atau pengoptimalan rute untuk masalah seperti Dijkstra dan Traveling Salesman Problem (TSP).

b. Desain Algoritma



Desain algoritma di atas mencakup berbagai operasi pada graf, termasuk algoritma Dijkstra untuk mencari jalur terpendek, pencarian jalur alternatif, dan solusi *Traveling Salesman Problem* (TSP). Proses dimulai dengan menampilkan menu utama yang menawarkan opsi untuk menampilkan graf, menjalankan algoritma *Dijkstra*, mencari jalur alternatif, menyelesaikan TSP, atau keluar dari program. Pada graf, hubungan antar-*node* ditampilkan dalam bentuk *adjacency list*. Algoritma Dijkstra mencari jalur terpendek dengan memprioritaskan *node* berdasarkan jarak terkecil dan mencetak hasilnya. Pencarian jalur alternatif dilakukan dengan memblokir *node* tertentu, menghapusnya dari *graf*, dan menjalankan Dijkstra kembali untuk menemukan jalur baru. Untuk TSP, program menghitung semua permutasi rute secara *brute-force*, mengevaluasi biaya setiap rute, dan menyimpan jalur

dengan biaya minimum sebagai solusi. Jika tidak ada jalur yang valid, solusi tidak ditemukan.

c. Implementasi Program

Dari rancangan yang sudah ada maka didapatkan implementasi program menggunakan bahasa pemrograman C++. Detail dari kode yang sudah sebagai berikut :

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <climits>
#include <algorithm>

using namespace std;

typedef pair<int, string> Pair; // Pair untuk priority queue (jarak, node)

class Graph {
private:
    unordered_map<string, vector<pair<string, int>>> adjList; // adjacency list

public:
    void addEdge(const string &u, const string &v, int weight) {
        adjList[u].push_back({v, weight});
        adjList[v].push_back({u, weight}); // Karena graf tidak berarah
    }

    void displayGraph() {
        cout << "Ilustrasi Graf:\n";
        for (const auto &node : adjList) {
            cout << "Node " << node.first << ": ";
            for (const auto &edge : node.second) {
                cout << " -> " << edge.first << "(" << edge.second << ")";
            }
```

```
            cout << "\n";
        }
    }

    void dijkstra(const string &start, const string &end) {
        if (adjList.find(start) == adjList.end() || adjList.find(end) == adjList.end()) {
            cout << "Gudang atau jalur tidak ada" << endl;
            return;
        }

        unordered_map<string, int> distances;
        unordered_map<string, string> parents;
        priority_queue<Pair, vector<Pair>, greater<Pair>> pq;

        // Inisialisasi jarak ke semua node dengan nilai maksimum
        for (const auto &node : adjList) {
            distances[node.first] = INT_MAX;
        }

        // Jarak ke node awal adalah 0
        distances[start] = 0;
        pq.push({0, start});

        while (!pq.empty()) {
            auto [currentDistance, currentNode] = pq.top();
            pq.pop();

            // Jika jarak yang sekarang lebih besar, lewati
            if (currentDistance > distances[currentNode])
                continue;

            // Periksa semua tetangga dari node saat ini
            for (const auto &neighbor : adjList[currentNode]) {
                const string &nextNode = neighbor.first;
                int weight = neighbor.second;

                // Hitung jarak baru
```

```
            int newDistance = currentDistance + weight;
            if (newDistance < distances[nextNode]) {
                distances[nextNode] = newDistance;
                parents[nextNode] = currentNode;
                pq.push({newDistance, nextNode});
            }
        }
    }

    // Cetak hasil jalur terpendek
    if (distances[end] == INT_MAX) {
        cout << "Gudang atau jalur tidak ada" << endl;
    } else {
        cout << "Jarak terpendek dari " << start << " ke " << end << " adalah " << distances[end] << endl;

        // Rekonstruksi jalur
        vector<string> path;
        for (string at = end; at != start; at = parents[at]) {
            path.push_back(at);
        }
        path.push_back(start);
        reverse(path.begin(), path.end());

        cout << "Jalur: ";
        for (const auto &node : path) {
            cout << node << (node == end ? "\n" : " -> ");
        }
    }
}

void findAlternateRoute(const string &start, const string &end, const string &blockedNode) {
    if (adjList.find(start) == adjList.end() || adjList.find(end) == adjList.end() || adjList.find(blockedNode) ==
adjList.end()) {
        cout << "Gudang atau jalur tidak ada" << endl;
        return;
    }
```

```
    cout << "\nMenghitung jalur alternatif dengan Gudang " << blockedNode << " Yang Tidak Bisa Dilewati:\n";

    // Menghapus semua edge yang terhubung ke blockedNode
    unordered_map<string, vector<pair<string, int>>> tempAdjList = adjList;
    for (auto &node : tempAdjList) {
        node.second.erase(remove_if(node.second.begin(), node.second.end(), [&blockedNode](pair<string, int> edge) {
            return edge.first == blockedNode;
        }), node.second.end());
    }
    tempAdjList.erase(blockedNode);

    // Menggunakan adjacency list sementara untuk Dijkstra
    unordered_map<string, int> distances;
    unordered_map<string, string> parents;
    priority_queue<Pair, vector<Pair>, greater<Pair>> pq;

    // Inisialisasi jarak ke semua node dengan nilai maksimum
    for (const auto &node : tempAdjList) {
        distances[node.first] = INT_MAX;
    }

    // Jarak ke node awal adalah 0
    distances[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [currentDistance, currentNode] = pq.top();
        pq.pop();

        // Jika jarak yang sekarang lebih besar, lewati
        if (currentDistance > distances[currentNode])
            continue;

        // Periksa semua tetangga dari node saat ini
        for (const auto &neighbor : tempAdjList[currentNode]) {
```

```
            const string &nextNode = neighbor.first;
            int weight = neighbor.second;

            // Hitung jarak baru
            int newDistance = currentDistance + weight;
            if (newDistance < distances[nextNode]) {
                distances[nextNode] = newDistance;
                parents[nextNode] = currentNode;
                pq.push({newDistance, nextNode});
            }
        }
    }

    // Cetak hasil jalur terpendek
    if (distances[end] == INT_MAX) {
        cout << "Gudang atau jalur tidak ada" << endl;
    } else {
        cout << "Jarak terpendek dari " << start << " ke " << end << " adalah " << distances[end] << endl;

        // Rekonstruksi jalur
        vector<string> path;
        for (string at = end; at != start; at = parents[at]) {
            path.push_back(at);
        }
        path.push_back(start);
        reverse(path.begin(), path.end());

        cout << "Jalur: ";
        for (const auto &node : path) {
            cout << node << (node == end ? "\n" : " -> ");
        }
    }
}

void solveTSP(const string &start) {
    vector<string> nodes;
    for (const auto &node : adjList) {
```

```
        nodes.push_back(node.first);
    }

    vector<string> bestPath;
    int minCost = INT_MAX;

    sort(nodes.begin(), nodes.end());

    do
    {
        if (nodes.front() != start)
            continue;

        int currentCost = 0;
        bool valid = true;

        for (size_t i = 0; i < nodes.size() - 1; ++i)
        {
            auto &current = nodes[i];
            auto &next = nodes[i + 1];

            bool found = false;
            for (const auto &neighbor : adjList[current])
            {
                if (neighbor.first == next)
                {
                    currentCost += neighbor.second;
                    found = true;
                    break;
                }
            }

            if (!found)
            {
                valid = false;
                break;
            }
```

```
        }

        if (valid)
        {
            auto &last = nodes.back();
            auto &first = nodes.front();

            bool found = false;
            for (const auto &neighbor : adjList[last])
            {
                if (neighbor.first == first)
                {
                    currentCost += neighbor.second;
                    found = true;
                    break;
                }
            }

            if (found && currentCost < minCost)
            {
                minCost = currentCost;
                bestPath = nodes;
            }
        }

    } while (next_permutation(nodes.begin(), nodes.end()));

    if (minCost == INT_MAX)
    {
        cout << "Gudang Tidak Ada\n";
    }
    else
    {
        cout << "Solusi TSP dengan biaya minimum: " << minCost << "\n";
        cout << "Jalur Yang Dilewati: \n";
        for (const auto &node : bestPath)
        {
```

```
            cout << node << " -> ";
        }
        cout << bestPath.front() << "\n";
    }
  }
};

int main() {
  Graph g;

  // Menambahkan edge sesuai dengan graf pada soal
  g.addEdge("A", "B", 10);
  g.addEdge("A", "C", 15);
  g.addEdge("B", "C", 5);
  g.addEdge("B", "D", 20);
  g.addEdge("C", "D", 10);
  g.addEdge("C", "E", 25);
  g.addEdge("D", "E", 5);
  g.addEdge("D", "F", 15);
  g.addEdge("E", "F", 10);

  int choice;
  string startNode, endNode, blockedNode;

  do {
    cout << "\nMenu:\n";
    cout << "1. Tampilkan Gudang Yang Ada\n";
    cout << "2. Cari Rute Menggunakan Djikstra\n";
    cout << "3. Temukan Jalur alternatif jika Gudang (Node) Tidak Bisa Dilewati\n";
    cout << "4. Mencari Rute Untuk Melewati Semua Gudang (TSP)\n";
    cout << "5. Keluar\n";
    cout << "Pilih opsi: ";
    cin >> choice;

    switch (choice) {
      case 1:
        g.displayGraph();
```

```
            break;
        case 2:
            cout << "Masukkan Gudang awal: ";
            cin >> startNode;
            cout << "Masukkan Gudang tujuan: ";
            cin >> endNode;
            g.dijkstra(startNode, endNode);
            break;
        case 3:
            cout << "Masukkan Gudang awal: ";
            cin >> startNode;
            cout << "Masukkan Gudang tujuan: ";
            cin >> endNode;
            cout << "Masukkan Gudang yang tidak bisa dilewati: ";
            cin >> blockedNode;
            g.findAlternateRoute(startNode, endNode, blockedNode);
            break;
        case 4:
            cout << "Masukkan Gudang awal: ";
            cin >> startNode;
            g.solveTSP(startNode);
            break;
        case 5:
            cout << "Keluar dari program.\n";
            break;
        default:
            cout << "Opsi tidak valid. Silakan coba lagi.\n";
    }
  } while (choice != 5);

  return 0;
}
```

Selain dalam bentuk *script* dalam bahasa pemrograman *C++*, dibuat juga *script* menggunakan *pseudocode* dengan isi sebagai berikut :

```
Kelas Graph:
   - adjList: adjacency list untuk graf

   metode addEdge(u, v, weight):
      - tambahkan edge antara node u dan v dengan weight yang diberikan

   metode displayGraph():
      - tampilkan adjacency list dari graf

   metode dijkstra(start, end):
      jika node start atau end tidak ada di adjList:
         cetak "Gudang atau jalur tidak ada"
      jika tidak:
         - inisialisasi jarak ke semua node sebagai infinity
         - set jarak ke node start sebagai 0
         - buat priority queue dengan (0, start)
         selama priority queue tidak kosong:
            - dapatkan node dengan jarak terkecil
            untuk setiap tetangga dari node saat ini:
               - hitung jarak baru
               jika jarak baru lebih kecil:
                  - perbarui jarak
                  - perbarui parent node
                  - tambahkan ke priority queue
         jika jarak ke node end adalah infinity:
            cetak "Gudang atau jalur tidak ada"
         jika tidak:
            - cetak jarak terpendek dari start ke end
            - rekonstruksi dan cetak jalur

   metode findAlternateRoute(start, end, blockedNode):
      jika node start, end, atau blockedNode tidak ada di adjList:
         cetak "Gudang atau jalur tidak ada"
      jika tidak:
         - buat adjacency list sementara tanpa blockedNode
         - inisialisasi jarak ke semua node sebagai infinity
         - set jarak ke node start sebagai 0
         - buat priority queue dengan (0, start)
         selama priority queue tidak kosong:
            - dapatkan node dengan jarak terkecil
            untuk setiap tetangga dari node saat ini:
```

```
                - hitung jarak baru
                jika jarak baru lebih kecil:
                    - perbarui jarak
                    - perbarui parent node
                    - tambahkan ke priority queue
            jika jarak ke node end adalah infinity:
                cetak "Gudang atau jalur tidak ada"
            jika tidak:
                - cetak jarak terpendek dari start ke end
                - rekonstruksi dan cetak jalur

    metode solveTSP(start):
        - buat daftar dari semua node
        - inisialisasi minCost sebagai infinity
        untuk setiap permutasi node:
            jika permutasi dimulai dengan start node:
                - hitung total biaya dari permutasi saat ini
                jika biaya lebih kecil dari minCost:
                    - perbarui minCost
                    - perbarui bestPath
        jika minCost adalah infinity:
            cetak "Gudang Tidak Ada"
        jika tidak:
            - cetak biaya minimum
            - cetak best path

metode main:
    - buat Graph g
    - tambahkan edge ke g
    - tampilkan menu:
        1. displayGraph()
        2. input startNode, endNode dan panggil dijkstra(startNode, endNode)
        3. input startNode, endNode, blockedNode dan panggil findAlternateRoute(startNode,
endNode, blockedNode)
        4. input startNode dan panggil solveTSP(startNode)
        5. keluar dari program
```

Output :

a. Menu Awal Program

```
Menu:
1. Tampilkan Gudang Yang Ada
2. Cari Rute Menggunakan Djikstra
3. Temukan Jalur alternatif jika Gudang (Node) Tidak Bisa Dilewati
4. Mencari Rute Untuk Melewati Semua Gudang (TSP)
5. Keluar
Pilih opsi:
```

b. Menampilkan Keseluruhan Gudang (Graf) yang Tersedia

```
Menu:
1. Tampilkan Gudang Yang Ada
2. Cari Rute Menggunakan Djikstra
3. Temukan Jalur alternatif jika Gudang (Node) Tidak Bisa Dilewati
4. Mencari Rute Untuk Melewati Semua Gudang (TSP)
5. Keluar
Pilih opsi: 1
Ilustrasi Graf:
Node F:  -> D(15) -> E(10)
Node E:  -> C(25) -> D(5) -> F(10)
Node B:  -> A(10) -> C(5) -> D(20)
Node D:  -> B(20) -> C(10) -> E(5) -> F(15)
Node C:  -> A(15) -> B(5) -> D(10) -> E(25)
Node A:  -> B(10) -> C(15)
```

c. Menjalankan Algoritma Graf untuk Mencari Jalur Tercepat

```
Menu:
1. Tampilkan Gudang Yang Ada
2. Cari Rute Menggunakan Djikstra
3. Temukan Jalur alternatif jika Gudang (Node) Tidak Bisa Dilewati
4. Mencari Rute Untuk Melewati Semua Gudang (TSP)
5. Keluar
Pilih opsi: 2
Masukkan Gudang awal: A
Masukkan Gudang tujuan: F
Jarak terpendek dari A ke F adalah 40
Jalur: A -> C -> D -> F
```

d. Mencari Jalur Alternatif Ketika Gudang Diblokir (Tidak Dapat Dilewati)

```
Menu:
1. Tampilkan Gudang Yang Ada
2. Cari Rute Menggunakan Djikstra
3. Temukan Jalur alternatif jika Gudang (Node) Tidak Bisa Dilewati
4. Mencari Rute Untuk Melewati Semua Gudang (TSP)
5. Keluar
Pilih opsi: 3
Masukkan Gudang awal: A
Masukkan Gudang tujuan: F
Masukkan Gudang yang tidak bisa dilewati: C

Menghitung jalur alternatif dengan Gudang C Yang Tidak Bisa Dilewati:
Jarak terpendek dari A ke F adalah 45
Jalur: A -> B -> D -> F
```

e. Solusi dari Traveling Salesman Problem (TSP)/Mencari cara agar dapat melewati semua gudang dengan rute paling efisien.

```
Menu:
1. Tampilkan Gudang Yang Ada
2. Cari Rute Menggunakan Djikstra
3. Temukan Jalur alternatif jika Gudang (Node) Tidak Bisa Dilewati
4. Mencari Rute Untuk Melewati Semua Gudang (TSP)
5. Keluar
Pilih opsi: 4
Masukkan Gudang awal: A
Solusi TSP dengan biaya minimum: 95
Jalur Yang Dilewati:
A -> B -> D -> F -> E -> C -> A
```

Penjelasan Kode :

Program ini mengimplementasikan struktur graf untuk menentukan rute pengiriman barang antar gudang. Program dibuat dengan beberapa fitur utama, seperti penggunaan algoritma Dijkstra untuk mencari jalur terpendek, perhitungan untuk mencari rute alternatif jika suatu gudang tidak dapat dilewati, dan solusi dari masalah *Traveling Salesman Problem* (TSP) untuk menemukan rute optimal yang dapat melewati semua gudang.

Graf dalam program ini direpresentasikan sebagai *adjacency list* menggunakan struktur *unordered_map*, di mana setiap gudang (node) dihubungkan dengan gudang lainnya yang terhubung, beserta dengan jarak (bobot) antar gudang.

Dari kode tersebut terdapat penjelasan dari fungsi fungsi utama yang ada

di dalam program yaitu :

- Sebagai bentuk representasi dan tampilan Graf yaitu Fungsi *addEdge()* digunakan untuk membuat *edge* dua arah antara dua node dengan bobot tertentu, yang menggambarkan jarak antar gudang. Metode *displayGraph()* digunakan untuk menampilkan hubungan atau keterkaitan antar gudang (*node),* sehingga fungsi ini dapat memberikan gambaran visual dari jaringan gudang yang saling terhubung.
- Algoritma Dijkstra Fungsi *dijkstra()* menghitung jalur terpendek antara dua gudang dengan menggunakan algoritma *Dijkstra*. Algoritma ini memanfaatkan *priority queue* untuk mengunjungi *node* dengan jarak terkecil terlebih dahulu. Ketika fungsi ini dijalankan maka jarak dari *node* awal ke setiap *node* lain secara terus diperbarui, serta informasi mengenai *node* induk disimpan untuk mencari atau merubah jalur terpendek. Hasil dari algoritma ini berupa total jarak serta rute yang dilalui, yang akan dicetak di layar.
- Perhitungan Rute Alternatif Fungsi *findAlternateRoute()* digunakan untuk mencari rute antara dua gudang dengan kondisi bahwa salah satu gudang tidak dapat dilewati. Fungsi ini memodifikasi *adjacency list* secara sementara untuk menghapus semua koneksi menuju gudang yang diblokir. Setelah itu, algoritma *Dijkstra* dijalankan kembali untuk mencari jalur terpendek tanpa melalui gudang yang tidak bisa dilewati.
- Masalah *Traveling Salesman* (TSP) Fungsi solveTSP() menyelesaikan masalah *Traveling Salesman* dengan mencoba semua cara dalam menyusun objek dengan memperhatikan urutan dari node yang ada, menghitung total biaya dari setiap rute, dan kembali dengan rute yang memiliki biaya terendah. Fungsi ini memastikan bahwa semua gudang dikunjungi tepat satu kali, sebelum kembali ke titik awal dengan biaya total yang seminimal mungkin.

C. Pembagian Tugas

Pada pengerjaan tugas besar ini proses pengerjaan dibagi menjadi :

Nama Anggota	Tugas	Prosentase
Satria Putra Dharma Prayudha	Pembuatan Menu Dari Program, Susunan Laporan, Membuat Fungsi "*findAlternateRoute*"	33,3%
Yogi Hafidh Maulana	Implementasi Algoritma Dijkstra, Membuat Ilustrasi Graf	33,3%
Ganesha Rahman Gibran	Membantu dalam isi Laporan , Membuat Desain Algoritma, Membuat Fungsi "*solveTSP*"	33,3%

D. Presentasi:

Link Video Presentasi

https://drive.google.com/file/d/1Y4xAdhsVUSJSuq0DIw6UDMYNSZOOGEHC/view?usp=drive_link