MODUL 9 TUGAS BESAR

Nafa Lutfia Atihrah Chandra (13221031)
Fathiya Amani Shabira (13221032)
Crysanta Caressa (13221033)
Petrus Nicolas Manurung (13221034)
Asisten: Emmanuella Pramudita Rumanti (13220031)
Kelompok: B05
EL2208-Praktikum Pemecahan Masalah dengan C
Laboratorium Dasar Teknik Elektro - Sekolah Teknik Elektro dan Informatika ITB

Abstrak

Praktikum Modul 9 Tugas Besar berisikan percobaan penyelesaian Travelling Salesman Problem dimana dicari rute terpendek atau jarak minimum dari data file external yang diberikan. Permasalahan TSP dapat diselesaikan menggunakan beberapa metode. Pada modul ini digunakan metode nearest neighbor dengan mempertimbangkan kelebihan dan kekurangannya dalam berbagai aspek. Laporan ini berisikan deskripsi permasalahan, analisis permasalahan dan flowchart algoritma yang digunakan. Hal-hal yang didapat pada modul ini, baik secara teoritis maupun praktikal, dapat dipahami dengan baik dan didapat kesesuaian antara data dengan referensi.

Kata kunci: Travelling Salesman Problem, Nearest Neighbor Algorithm, Pemrograman Bahasa C

1. PENDAHULUAN

Permasalahan yang akan diselesaikan oleh kelompok kami adalah mencari rute pelayaran paling efisien, total jarak yang ditempuh dan pelabuhan yang tidak dapat dikunjungi karena letaknya yang terlalu jauh. Inputnya berupa file eksternal yang berisikan nama pelabuhan serta posisi lintang dan posisi bujur dari pelabuhan tersebut. Sedangkan output yang diharapkan berupa semua pelabuhan yang dilalui dalam upaya mencapai rute pelayaran optimal, total jarak yang ditempuh dan nama-nama pelabuhan yang tidak dapat dikunjungi. Secara garis besar, kelompok praktikan mendekomposisikan masalah menjadi 3 bagian antara lain:

- ***Input***

Ketika user memasukkan nama file eksternal sebagai input, program akan memeriksa keberadaan dan isi file tersebut. Jika file tidak ditemukan atau kosong, program akan menghasilkan pesan error. Namun, jika file eksternal berhasil ditemukan, langkah awal yang dilakukan adalah membaca file external tersebut yang berisi informasi mengenai nama pelabuhan, posisi lintang dan posisi bujur dari pelabuhan tersebut menggunakan fungsi filetoArray(). Selanjutnya, program menyimpan data pelabuhan tersebut dalam array struct Pelabuhan. Program kemudian akan membuat matriks jarak antar pelabuhan menggunakan data pelabuhan yang telah disimpan dalam array menggunakan fungsi arraytoMatrix().

- **Pemrosesan Algoritma TSP**

Langkah selanjutnya adalah program akan mencari pelabuhan pertama yang akan dikunjungi menggunakan fungsi harbor_check(). Kemudian, program memanggil fungsi tsp() dengan pelabuhan pertama sebagai titik awal. Fungsi tsp() mencari pelabuhan terdekat yang belum dikunjungi, memanggil dirinya sendiri dengan pelabuhan tersebut sebagai titik awal, dan menambahkan jarak antara pelabuhan yang dikunjungi ke variabel jarak. Proses ini berulang sampai semua pelabuhan telah dikunjungi.

- ***Output***

Output yang dihasilkan bergantung pada input nama file yang diberikan pada user. Jika file input memiliki format nama yang salah (tidak sesuai dengan spesifikasi), maka program akan mengeluarkan pesan, "Tidak ada file". Jika file input kosong, maka akan dikeluarkan pesan, "Error: File empty". Selain dari kedua keadaan di atas, program akan mengeluarkan output semua pelabuhan yang dilalui dalam upaya mencapai rute pelayaran optimal, total jarak yang ditempuh dan nama-nama pelabuhan yang tidak dapat dikunjungi.

2. STUDI PUSTAKA

2.1 TRAVELLING SALESMAN PROBLEM

Travelling Salesman Problem (TSP) adalah pencarian rute terpendek atau jarak minimum oleh seorang salesman dari suatu kota ke n-kota tepat satu kali dan kembali ke kota awal keberangkatan.[1] Dengan menggunakan rumus,

$$\frac{(n-1)!}{2}$$

diperoleh banyaknya jalur yang dapat ditempuh dari n-kota tersebut. Namun, untuk nilai n yang sangat besar, algoritma ini sangat tidak efisien. Secara teori, tidak ada algoritma yang dapat mencari solusi paling efisien dari masalah ini, namun ada beberapa teknik yang telah dikembangkan untuk menyelesaikan TSP antara lain:

1. *Brute-force:* Metode ini melibatkan pengecekan semua kemungkinan rute yang dapat diambil dan memilih rute terpendek. Namun, metode ini hanya dapat digunakan untuk jumlah kota yang kecil, karena kompleksitasnya yang tinggi.
2. *Nearest neighbor algorithm*: Metode ini bekerja dengan memilih kota yang paling dekat dengan kota saat ini sebagai kota berikutnya yang harus dikunjungi. Algoritma ini dapat menghasilkan solusi yang baik, tetapi tidak selalu menghasilkan solusi yang optimal. Algoritma akan dipakai dalam program yang memiliki metode yang dapat dijelaskan seperti dibawah:
   a. Dimulai dari *root* yang sebarang kemudian dipilih 'tujuan' paling dekat dari awal dimulainya rute.
   b. Setelah rute pertama ditentukan, berdasarkan sisa-sisa titik tempuh sekelilingnya, dipilih dari nilai/*weight path* paling kecil.
   c. Ulangi langkah b hingga seluruh rute terdiri atas semua titik dari n-kota.
3. *Christofides algorithm*: Algoritma ini menggabungkan strategi greedy dan teknik pembentukan pohon minimum (minimum spanning tree) untuk menyelesaikan TSP. Algoritma ini dapat menghasilkan solusi yang optimal dengan waktu yang efisien.
4. *Genetic algorithm*: Metode ini memodelkan TSP sebagai masalah optimasi kombinatorial dan menggunakan teknik evolusi untuk menyelesaikan masalah. Algoritma ini dapat menghasilkan solusi yang baik dengan waktu yang efisien.
5. *Simulated annealing*: Metode ini menggunakan teknik optimasi global dengan mengacu pada proses fisik simulasi annealing dalam pemrosesan logam. Algoritma ini dapat menemukan solusi yang baik dalam waktu yang wajar. [2]

2.2 Nearest Neighbour Algorithm

K-Nearest Neighbor (k-NN atau KNN) adalah sebuah metode untuk melakukan klasifikasi terhadap objek berdasarkan data pembelajaran (neighbor) yang jaraknya paling dekat dengan objek tersebut. Dekat atau jauhnya neighbor biasanya dihitung berdasarkan jarak Euclidean. diperlukan suatu sistem klasifikasi sebagai sebuah sistem yang mampu mencari informasi.[3]

2.3 Greedy Algorithm

Algoritma Greedy adalah algoritma yang memecahkan persoalan secara langkah per langkah (*step by step*). Pada setiap langkahnya, algoritma memilih pilihan terbaik yang dapat diperoleh pada saat itu tanpa memperhatikan konsekuensi kedepannya (prinsip "*take what you can get now*!") dan "berharap" bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global. Algoritma greedy digunakan untuk memecahkan persoalan optimasi, yaitu persoalan mencari solusi optimal (maksimisasi atau minimisasi).

2.4 Rumus Haversine

Rumus Haversine adalah rumus matematika yang digunakan untuk menghitung jarak antara dua titik pada permukaan bola (seperti Bumi) menggunakan koordinat lintang dan bujur. Rumus ini walaupun hanya sebuah aproksimasi, dapat memberikan hasil yang akurat dalam menghitung jarak antara dua titik pada permukaan bola tsb. Rumus Haversine dapat dijabarkan sebagai berikut,

$$a = sin^2(\Delta\varphi/2) + cos\varphi_1 . cos\varphi_2 . sin^2(\Delta\lambda/2)$$

$$c = 2 . atan2(\sqrt{a}, \sqrt{(1-a)})$$

$$d = R . c$$

dimana,

φ = *latitude*,
λ = *longitude*,
R = jari-jari Bumi (6,371 km).

3. METODOLOGI

3.1 MENDEFINISIKAN RUANG LINGKUP MASALAH

Gambar 3-1 Diagram Langkah-langkah Pengerjaan Mendefinisikan Ruang Lingkup Masalah

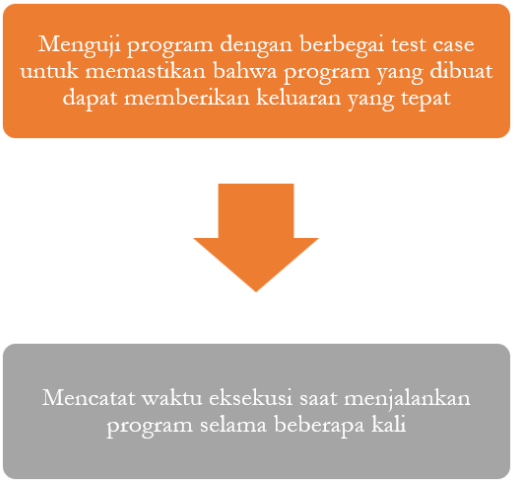
3.2 MERANCANG *SOFTWARE*

Gambar 3-2 Diagram Langkah-langkah Pengerjaan Merancang *Software*

3.3 IMPLEMENTASI RANCANGAN

Gambar 3-3 Diagram Langkah-langkah Pengerjaan Implementasi Rancangan

3.4 PENGUJIAN



Gambar 3-4 Diagram Langkah-langkah Pengerjaan Pengujian

4. HASIL DAN ANALISIS

4.1 RUANG LINGKUP MASALAH

Dalam mengerjakan program yang akan dibuat, terdapat keterbatasan dan kebebasan spesifikasi seperti:

1. Jarak tempuh maksimal kargo kapal api standar 2500 km,

   Hasil pada program, kode akan terus menambah jarak terdekat antar pelabuhan *neighbour (adjacent)* pada *map* dengan perbandingan iterasi *if* sederhana dengan variabel **min** yang akan diubah menjadi jarak dan **nh** menjadi nomor pelabuhan tersebut dan jika jarak tempuh maksimal

sudah tercapai (atau >2500 km) maka program akan mengembalikan pelabuhan selanjutnya kembali ke pelabuhan awal dan jaraknya tidak ditambahkan ke jarak total.

2. Efisiensi algoritma yang dipilih berdasarkan *Travelling Salesman Problem* menggunakan algoritma *Nearest-Neighbor Method*,

   Akumulasi *path* akhir dapat ditentukan menggunakan fungsi implementasi **tsp** pada program yang menggunakan pendekatan rekursif. Dipetik dari 2.1.2 fungsi akan lebih baik apabila menggunakan metode *nearest neighbour*. Untuk menghindari pelabuhan yang sama dilewati lebih dari sekali, maka fungsi akan men-*flag* pelabuhan *current* sebagai sudah dikunjungi (= 1) dan dicetak ke keluaran program untuk dijadikan *history* rute.

3. Perhitungan aproksimasi jarak di dunia avatar menggunakan rumus *haversine*.

   Kode tersebut menggunakan rumus *Haversine* untuk menghitung jarak antara dua titik koordinat berdasarkan lintang dan bujur yang diberikan. Namun, rumus *Haversine* hanya memberikan aproksimasi jarak karena pada naskah ini rujukan bumi hanyalah sebuah fiktif belaka/imajinatif). Oleh karena itu, perhitungan jarak yang dihasilkan oleh kode ini hanya bersifat aproksimasi dan tidak sepenuhnya akurat.

Sehingga, berdasarkan kasus-kasus tersebut dihasilkan kesimpulan dengan pembulatan asumsi terhadap:

1. *Input* dan *Output* Program.

   Program tidak menggunakan *input* dari pengguna selain nama file, sehingga tidak ada interaksi langsung pengaruh pengguna yang dapat mempengaruhi *output*. Namun, jika file yang digunakan berbeda, maka *output* akan berbeda karena isi file tersebut berbeda. Hal ini dikarenakan pada program, keluaran akan sama setiap kali program dijalankan dengan data file yang sama karena tidak ada penggunaan nilai acak *(random)* dalam program tersebut.

   Kemudian, diberikan *testcase* baru yang memastikan nama dari file (pelabuhan.csv) sama dan memiliki isi data yang sesuai. Jika tidak, maka akan memunculkan *error message* dibanding pengolahan data seharusnya. Hasil implementasi dapat diperoleh pada Bagian 4.4.

2. Awal mulai rute pelayaran yang *random*/bebas (*arbitrary root*).

   Program memulai rute pelayaran yang ditentukan secara acak karena algoritma yang digunakan untuk menentukan solusi rute pelayaran yang optimal adalah *nearest-neighbor*. Algoritma dimulai dari titik awal yang dipilih secara acak, kemudian memilih pelabuhan terdekat yang belum dikunjungi sebagai pelabuhan selanjutnya dan seterusnya hingga semua pelabuhan dikunjungi (dengan total jarak tempuh <2500 km). Titik awal ditentukan oleh fungsi **harbor_check** yang juga memilih secara acak satu pelabuhan sebagai titik awal jika pelabuhan awal belum ditentukan. Oleh karena itu, rute pelayaran pada program dilakukan secara acak.

3. Pengolahan besar data hanya pada 20 (titik) pelabuhan.

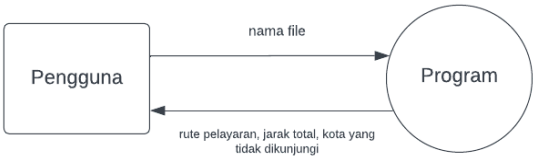
   Pada bagian awal telah diinisialisasi **maxpelabuhan** yang menentukan ukuran dari *array* yang digunakan untuk menyimpan data pelabuhan(titik kota) dan matriks jarak. Apabila file yang diolah memiliki <20 data n-kota maka program akan menampilkan data berdasarkan jumlah data pada file sehingga maksimum hanya pada <20 (asumsi tidak ada kota yang dilewatkan). Namun, apabila data memiliki besar >20 maka, rute yang diperoleh hanya sampai 20 titik (kota) pelabuhan dan sisa akan otomatis masuk ke dalam *list* pada *output* kota yang tidak dilewati.

Dihasilkan kasus terakhir yang diperoleh setelah program dibuat yaitu dua Desa (Desa Hira'a Tahiti dan Makapu Oahu) yang tidak bisa muncul pada keluaran sebagai salah satu rute dari pelayaran Negara Api. Pada keluaran kode tersebut terdapat kota yang tidak dilalui karena pelabuhan tersebut tidak dianggap optimal untuk dilewati pada solusi rute pelayaran terpendek. Hal ini terjadi karena algoritma yang digunakan pada fungsi **tsp** adalah algoritma Nearest Neighbor (NN) yang hanya mengambil titik terdekat dari pelabuhan yang telah dikunjungi sebagai pelabuhan selanjutnya. Sehingga pada solusi rute pelayaran yang dihasilkan, tidak semua

pelabuhan dilewati. Setelah itu, pada fungsi **skipped_harbor** akan diperoleh daftar pelabuhan yang tidak dilewati.

4.2 RANCANGAN

Program dirancang untuk mencari dan menunjukkan rute pelayaran yang optimal antar pelabuhan serta catatan tambahan tentang pelabuhan yang tidak dilewati. Pengguna dapat berinteraksi melalui masukan file yang berisi data pelabuhan beserta titik koordinatnya yang akan diproses. Setelah itu, program akan melakukan penghitungan jarak antar pelabuhan dan menampilkan matriks jarak antar pelabuhan yang sudah di define sebesar 20x20 (variabel **maxpelabuhan**). Kemudian, program akan menentukan rute pelayaran yang optimal sebagai solusi dari *travelling salesman problem* menggunakan algoritma *nearest-neighbor* dan mencetak rute pelayaran tersebut beserta jarak total rute pelayaran dan pelabuhan yang tidak dilewati pada rute pelayaran tersebut. Diagram dari Interaksi tersebut dapat diperjelas dengan Gambar 4-1.

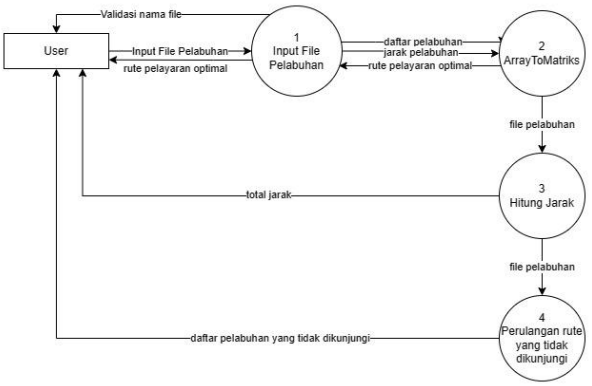


Gambar 4-1 DFD Level 0

Penyelesaian *Travelling Salesman Problem* sangat baik diselesaikan menggunakan Algoritma *Nearest-Neighbor* seperti yang sudah dijelaskan pada Bagian 2.1 poin 2 serta Bagian 3.1 poin 2 pertama.

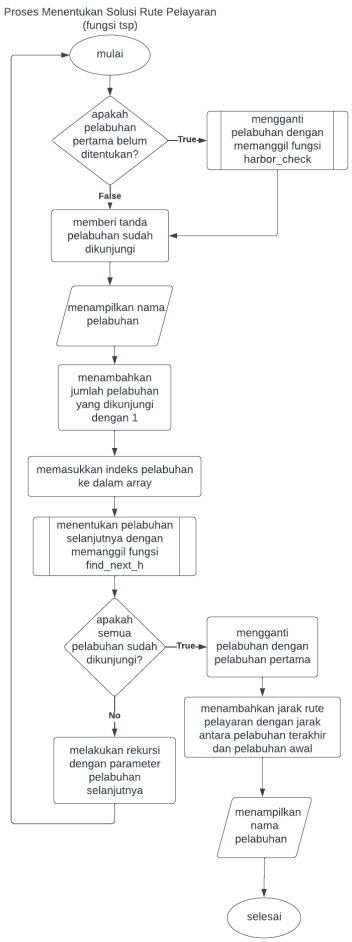
Program akan membaca file masukan dan memasukkan isinya ke dalam *array* yang sudah di definisikan dalam *struct* **pelabuhan.** Kemudian, dari data pada file pelabuhan.txt akan dimasukkan ke dalam matriks yang menyimpan jarak antar setiap pelabuhan dengan menggunakan rumus Haversine pada setiap Pelabuhan (berdasarkan koordinat lintang dan bujurnya). Lalu, program akan mencari rute pelayaran menggunakan algoritma *Travelling Salesman Problem* dengan fungsi **tsp** untuk melakukan pemanggilan fungsi rekursif **find_next_h** untuk menentukan pelabuhan selanjutnya tersebut yang akan ditandai sebagai sudah dikunjungi dengan memberikan nilai 1 pada **visited_harbor**. Fungsi **tsp** akan terus memanggil fungsi **find_next_h** hingga seluruh pelabuhan telah dikunjungi dan kembali ke pelabuhan awal. Setelah itu, dari **jarak** total tempuh fungsi **tsp** akan dihitung dan dicetak pada *output* akhir (selama fungsinya berjalan). Kemudian, di *output* pun akan dicetak deretan pelabuhan yang tidak dilewati oleh negara api dengan tanda nilai 0 pada **visited_neighbor** pada fungsi **skipped_harbor**.

Serangkaian tahap alur program dengan Algoritma *Travelling Salesman* dan *Nearest-Neighbor* dapat digambarkan dengan DVD Level 1 pada Gambar 4-2.



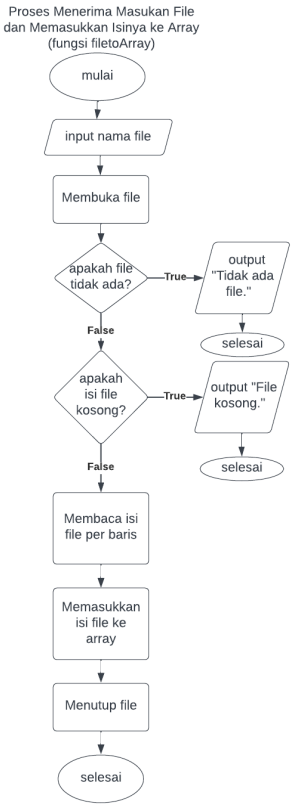
Gambar 4-2 DFD Level 1

- Fungsi Komplementer
  1. Fungsi **tsp()**



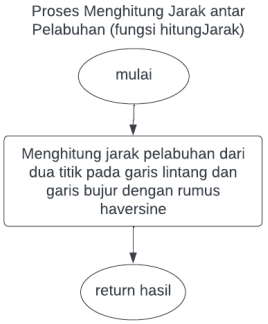
Gambar 4-3 *Flowchart* fungsi tsp()

2. Fungsi **filetoArray()**



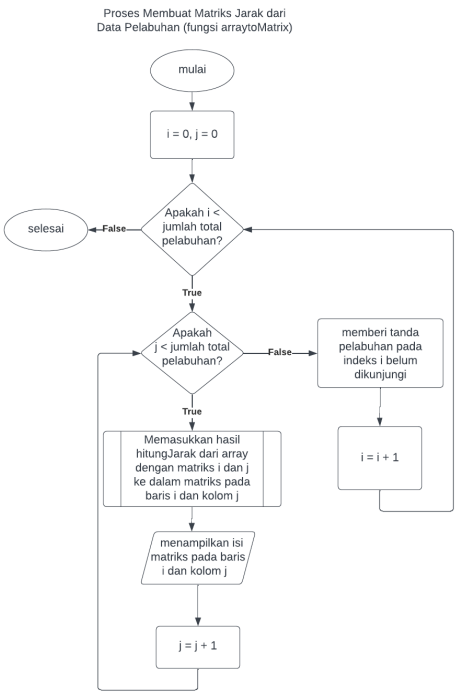
Gambar 4-4 *Flowchart* fungsi filetoArray()

3. Fungsi **hitungJarak()**



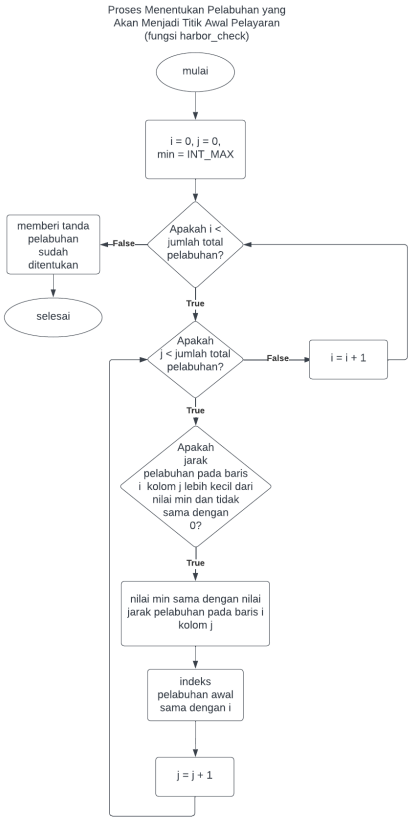
Gambar 4-5 *Flowchart* fungsi hitungJarak()

4. Fungsi **arraytoMatrix()**



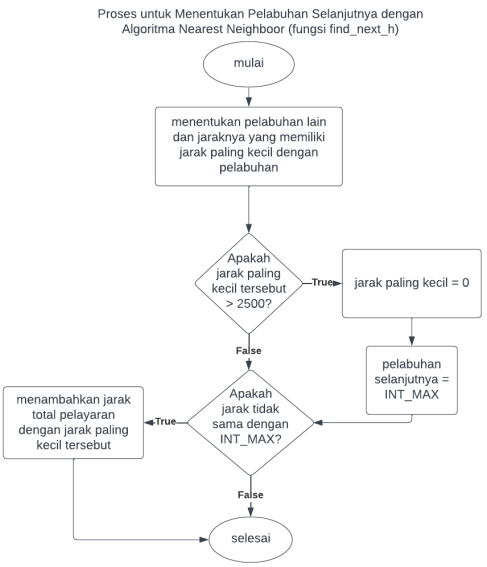
Gambar 4-6 *Flowchart* fungsi arraytoMatrix()

5. Fungsi **harbor_check()**



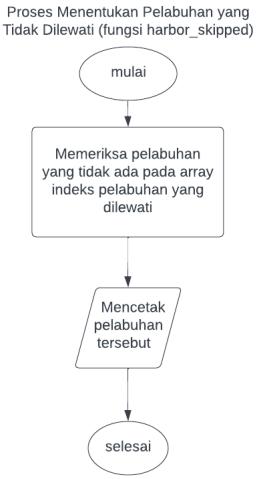
Gambar 4-7 *Flowchart* fungsi harbor_check()

6. Fungsi **find_next_h()**



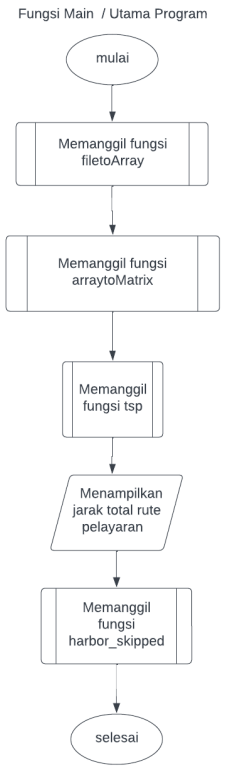
Gambar 4-8 *Flowchart* fungsi find_next_h()

7. Fungsi **harbor_skipped()**



Gambar 4-9 *Flowchart* fungsi harbor_skipped()

- Fungsi Algoritma Utama



Gambar 4-10 *Flowchart* fungsi main()

Tabel 4-1 Tabel Rencana Pembagian Tugas

Task	Pembagian Tugas
fungsi filetoArray	Implementer: 13221033 Tester: -13221031 -13221032 -13221033 -13221034
fungsi harbor_check, tsp, find_next_h, skipped harbor	Implementer: 13221032 -13221031 -13221032 -13221033 -13221034
fungsi main, arraytoMatrix, hitungJarak, tsp, find_next_h	Implementer: 13221034 Tester: -13221031 -13221032 -13221033 -13221034

4.3 IMPLEMENTASI

4.3.1. IMPLEMENTASI FUNGSI

```
struct Pelabuhan
{
   char nama[50];
   double lintang;
   double bujur;
};
```

Program diawali dengan inisialisasi structure berdasarkan struktur data yang akan diolah (*.csv) yang berisikan nama (titik) kota rute pelayaran, derajat lintang dan derajat bujur pada map dunia avatar.

```
int main()
{
   filetoArray(pelabuhan);
   printf("\n\nMatriks jarak antar
pelabuhan : \n");
   arraytoMatrix(pelabuhan, jrk_pelabuhan);

   printf("\n\nRute Pelayaran Optimal:\n");
   tsp(0);
   printf("\n\nJarak Total Rute Pelayaran:
");
   printf("%.3f km\n", jarak);

   printf("\n\nKota yang Tidak
Dilewati:\n");
   skipped_harbor();
   return 0;
}
```

Algoritma Fungsi Utama berisikan pemanggilan *predefined process*/fungsi operasional komplementari pada keseluruhan program. Berdasarkan rancangan, data file yang akan dibaca akan dimasukkan ke dalam *array* (**filetoArray).** Langkah selanjutnya adalah untuk men-*trace* jejak dari rute pelayaran yang akan, telah, dan yang tidak ditempuh pada matriks (**arraytoMatrix**).

Lalu, untuk keluaran yang sesuai naskah soal, yaitu pemanggilan fungsi **tsp** untuk menampilkan rute berurut pelabuhan yang bisa dikunjungi, memanggil variabel hasil fungsi **hitungJarak** yang menunjukkan total jarak tempuh, serta memanggil fungsi **harbor_skipped** untuk menampilkan *list* pelabuhan mana saja yang dilewati. Apabila dibandingkan dengan rancangan *flowchart* fungsi **main()** dari Bagian 4.2, implementasi kode cukup akurat.

```
void filetoArray(struct Pelabuhan
array[maxpelabuhan]){
   FILE *fp;
   char line[maxline];
   char namafile[100];
   char *token;
   int count = 0;

   printf("Masukkan nama file: ");
   scanf("%s", namafile);
   fp = fopen(namafile, "r");
   // Jika file tidak ada
   if (fp == NULL)
   {
       printf("\nTidak ada file.");
       exit(0);
   }

   // Jika file kosong
   int c = fgetc(fp);
   if (c == EOF){
       printf("\nFile kosong.");
       exit(0);
   }
   ungetc(c,fp);

   // Membaca isi file dan memasukkan isi
file ke dalam array
   fgets(line, maxline,fp);
   while (fgets(line, maxline, fp))
   {
       token = strtok(line, ",");
       strcpy(array[count].nama, token);
       token = strtok(NULL, ",");
       array[count].lintang = atof(token);
       token = strtok(NULL, ",");
       array[count].bujur = atof(token);
       count++;
   }

   fclose(fp);
}
```

Proses komplementari yang pertama adalah fungsi **filetoArray**. Prosedur ini akan menerima masukan sebuah string yang merupakan nama file berisi data-data pelabuhan (nama, lintang, bujur). Prosedur kemudian akan memeriksa apakah nama file yang dimasukkan tersedia dan bukan merupakan file kosong. Jika ternyata tidak terdapat file dengan nama yang dimasukkan atau file merupakan file kosong maka program akan menampilkan pesan error dan program akan otomatis berhenti. Jika file tersedia dan berisi data pelabuhan, maka program akan membaca data kemudian menyimpan masing-masing data pelabuhan ke dalam struct Pelabuhan yang kemudian akan disimpan ke dalam array of struct.

```
void arraytoMatrix(struct Pelabuhan
array[maxpelabuhan], double
matrix[maxpelabuhan][maxpelabuhan]){
   for(int i = 0; i < maxpelabuhan; i++){
       for(int j = 0; j < maxpelabuhan;
j++){
           matrix[i][j] =
hitungJarak(array[i], array[j]);
           printf("%.1f ", matrix[i][j]);
       }
       printf("\n");
       visited_harbor[i]=0;
   }
}
```

Prosedur **arraytoMatrix** merupakan proses untuk membuat matriks yang berisi jarak antar pelabuhan yang akan digunakan untuk menentukan rute dengan jarak terkecil. Prosedur ini akan membaca data lintang dan bujur dari masing-masing pelabuhan dari array yang sudah dibuat sebelumnya kemudian menggunakan fungsi hitungJarak untuk membantu menghitung jarak kedua pelabuhan berdasarkan lintang dan bujurnya.

```
double hitungJarak(struct Pelabuhan
pelabuhan1, struct Pelabuhan pelabuhan2){
   double hasil;
   double a =
pow(sin(convert_RadiantoDegree(pelabuhan1.l
intang-pelabuhan2.lintang)/2),2) +
cos(convert_RadiantoDegree(pelabuhan1.linta
ng))*cos(convert_RadiantoDegree(pelabuhan2.
lintang))*pow(sin(convert_RadiantoDegree(pe
labuhan1.bujur-pelabuhan2.bujur)/2),2);
   double c = 2*atan2(sqrt(a), sqrt(1-a));
   hasil = radius*c;
   return hasil;
}
```

Fungsi **hitungJarak** merupakan fungsi yang menghitung jarak antar pelabuhan menggunakan rumus haversine yang dapat diimplementasi menggunakan perluasan *library C* <math.h> sesuai

dengan rumus pada Bagian 2.4 yang diolah dari data file masukan **pelabuhan.csv** pada garis lintang dan garis bujur. **hasil** yang akan di-*return* berupa tipe data *double*.

```
int harbor_check(int h){
   double min = INT_MAX;
   for (int i = 0; i < maxpelabuhan; i++){
       for(int j = 0; j < maxpelabuhan;
j++){
           //Menentukan jarak yang paling
kecil dari matriks jarak
           if (jrk_pelabuhan[i][j] < min &&
jrk_pelabuhan[i][j]!= 0){
               min = jrk_pelabuhan[i][j];
               h = i;
           }
       }
   }
   first_harbor_flag = 0;
   first_harbor = h;
}
```

Fungsi **harbor_check** digunakan untuk menentukan pelabuhan mana yang akan menjadi pelabuhan pertama dalam rute yang akan ditentukan. Fungsi ini akan memeriksa jarak antar pelabuhan terkecil dan bukan nol yang terdapat dalam matriks kemudian menyimpan indeks dari pelabuhan tersebut ke variabel first_harbor.

```
void tsp(int h){
   int next_h;
   //Menentukan pelabuhan titik awal
pelayaran
   if(first_harbor_flag == 1){
       h = harbor_check(h);
   }
   visited_harbor[h] = 1;

   // Mencetak nama pelabuhan yang dilewati
   printf("%s --> ", pelabuhan[h].nama);

   //Menambahkan indeks pelabuhan yang
dilewati ke dalam array harbor_num
   harbor_total++;
   harbor_num[harbor_total] = h;

   //Menentukan pelabuhan selanjutnya
   next_h = find_next_h(h);

   // Bagian basis jika semua pelabuhan
sudah dilewati oleh fungsi tsp
   if(next_h == INT_MAX){
       next_h = first_harbor;
       jarak += jrk_pelabuhan[h][next_h];
       printf("%s",
pelabuhan[next_h].nama);
       return;
   }

   //Bagian rekursi dengan memanggil fungsi
tsp dengan parameter pelabuhan selanjutnya
   tsp(next_h);

}
```

Prosedur **tsp** merupakan proses rekursif untuk menentukan rute terpendek berdasarkan matriks jarak antar pelabuhan. Pertama, prosedur menentukan pelabuhan yang akan menjadi titik awal menggunakan fungsi harbor_check kemudian prosedur akan menandai bahwa pelabuhan tersebut telah ditandai dengan menyimpan nilai 1 pada array visited_harbor. Setelah itu prosedur akan menampilkan keluaran nama pelabuhan kemudian mencari pelabuhan berikutnya dengan bantuan fungsi find_next_h. Jika fungsi find_next_h mengembalikan nilai INT_MAX, maka semua pelabuhan telah dikunjungi dan prosedur akan kembali pada pelabuhan pertama serta menjumlahkan jarak pelabuhan terakhir dengan pelabuhan pertama dengan variabel jarak. Jika fungsi find_next_h tidak mengembalikan nilai INT_MAX, prosedur akan memanggil kembali proses tsp.

```
int find_next_h(int h){
   int nh;
   double min = INT_MAX, temp_jrk=0;

   for(int i=0; i< maxpelabuhan; i++){
       //Menentukan pelabuhan dan jaraknya
yang memiliki jarak paling kecil dengan
pelabuhan h
       if(visited_harbor[i] == 0 &&
jrk_pelabuhan[h][i] != 0 &&
jrk_pelabuhan[h][i] < min){
           temp_jrk = jrk_pelabuhan[h][i];
           min = jrk_pelabuhan[h][i];
```

```
            nh = i;
        }
    }

    // Jika jarak minimal lebih besar dari
2500, jarak minimal dibuat 0 agar tidak
ditambahkan ke jarak total dan pelabuhan
selanjutnya kembali ke awal
    if (min > 2500){
        min = 0;
        nh = INT_MAX;
    }

    if(temp_jrk != INT_MAX){
        //Menambahkan jarak minimal
pelabuhan ke dalam total jarak rute
pelayaran
        jarak += min;
    }
    return nh;
}
```

Fungsi **find_next_h** merupakan fungsi untuk mencari pelabuhan selanjutnya yang akan dikunjungi dalam rute. Fungsi ini menerima argumen h berupa integer yang merujuk pada indeks pelabuhan sebelumnya kemudian fungsi akan mengecek pelabuhan mana yang belum dikunjungi dan memiliki jarak terkecil dengan pelabuhan sebelumnya. Setelah itu fungsi akan memeriksa apakah jarak antar pelabuhan lebih kecil dari 2500, jika jarak antar pelabuhan lebih dari 2500 maka fungsi akan mengembalikan nilai INT_MAX yang berarti kembali ke pelabuhan awal. Jika jarak antar pelabuhan lebih kecil dari 2500 maka fungsi akan menambahkan jarak ke variabel jarak dan akan mengembalikan integer yang merujuk pada indeks pelabuhan berikutnya. Jika semua pelabuhan sudah dikunjungi, fungsi akan mengembalikan nilai INT_MAX.

```
void skipped_harbor(){
    bool not_route = false;
    int num = 1;
    for(int i = 0; i < maxpelabuhan; i++){
        for(int j = 0; j <= harbor_total;
j++){
            if(i == harbor_num[j]){
                not_route = true;
            }
        }
        //Mencetak pelabuhan yang indeksnya
tidak ada di dalam array_num
        if (!not_route){
            printf("%d. %s\n", num,
pelabuhan[i].nama);
            num++;
        }
        not_route = false;
    }
}
```

Prosedur **skipped_harbor** merupakan proses untuk menampilkan pelabuhan yang tidak dikunjungi karena jarak melebihi 2500. Prosedur akan memeriksa apakah sebuah pelabuhan termasuk pada rute atau tidak berdasarkan data yang disimpan pada array harbor_num kemudian akan menampilkan keluaran pelabuhan mana saja yang tidak dapat dikunjungi.

4.3.2. Analisis Kompleksitas Ruang dan Waktu

Terdapat beberapa fungsi/prosedur yang digunakan pada program namun secara garis besar program berfokus pada prosedur tsp() sehingga akan dianalisis kompleksitas prosedur tsp sebagai kompleksitas waktu abstraksi program. Prosedur tsp merupakan proses rekursif sehingga memiliki kompleksitas waktu abstraksi O(n) namun didalam prosedur tsp dilakukan pemanggilan fungsi find_next_harbor yang memiliki proses iterasi bersarang dengan kompleksitas waktu $O(n^2)$ sehingga prosedur tsp memiliki kompleksitas waktu $O(n^2)$ dengan n jumlah pelabuhan yang diproses.

Pada program, data-data pelabuhan diproses terlebih dahulu sehingga didapat matriks yang berisi jarak antar pelabuhan sehingga ruang yang digunakan dalam program adalah sebesar ruang yang digunakan untuk menyimpan matriks dan beberapa variabel lainnya. Sehingga program memiliki kompleksitas ruang $O(n^2)$ dengan n jumlah pelabuhan.

4.4 Pengujian

4.4.1. Test Case

- Test Case 1

Kasus pertama adalah kasus ketika pengguna memberikan masukan yang tidak valid atau file tidak tersedia.

```
Masukkan nama file: stasiun.csv

Tidak ada file.
```

- Test Case 2

Kasus berikutnya adalah kasus ketika file tersedia namun tidak menyimpan data pelabuhan (file kosong)

```
Masukkan nama file: file1.csv

File kosong.
```

- Test Case 3

Kasus berikutnya adalah kasus ketika file tersedia dan berisi data-data pelabuhan

```
Masukkan nama file: pelabuhan.csv

Matriks jarak antar pelabuhan :
0.0 1653.9 2882.9 2889.0 431.8 3705.4 2747.8 1161.1 1663.0 2229.5 2533.6 3248.2 3819.7 3713.3 2025.6 1699.9 2873.2 2303.2 2486.7 3109.8
1653.9 0.0 2727.3 1323.3 1614.4 3808.7 2437.9 570.7 9.1 1885.9 2569.1 2429.7 2632.4 2309.2 387.8 559.2 1783.0 2072.9 2224.6 2684.0
2882.9 2727.3 0.0 2641.3 2456.7 6364.0 5098.6 2434.9 2729.5 841.8 402.7 1148.6 2050.6 2494.5 2934.7 2172.1 1605.4 665.2 502.7 468.2
2889.0 1323.3 2641.3 0.0 2730.5 4792.1 3366.7 1728.0 1315.5 1962.7 2687.1 1804.7 1562.7 1060.5 1091.0 1215.3 1094.6 2157.3 2216.9 2371.4
431.8 1614.4 2456.7 2730.5 0.0 4112.7 3089.5 1056.1 1623.3 1832.4 2103.2 2881.1 3508.8 3465.2 2001.7 1515.3 2567.8 1891.8 2073.4 2698.9
3705.4 3808.7 6364.0 4792.1 4112.7 0.0 1425.3 3945.5 3810.7 5555.2 6090.6 6238.4 6332.1 5842.5 3801.8 4339.7 5574.7 5705.3 5883.3 6430.8
2747.8 2437.9 5098.6 3366.7 3089.5 1425.3 0.0 2666.2 2438.6 4265.7 4869.9 4856.3 4909.4 4417.9 2391.3 2987.4 4174.3 4433.3 4602.6 5109.5
1161.1 570.7 2434.9 1728.0 1056.1 3945.5 2666.2 0.0 579.2 1611.4 2208.4 2391.5 2788.6 2593.1 956.9 567.5 1860.9 1770.3 1943.6 2487.7
1663.0 9.1 2729.5 1315.5 1623.3 3810.7 2438.6 579.2 0.0 1888.1 2572.5 2427.8 2627.5 2302.6 379.0 560.5 1779.7 2075.4 2226.7 2684.6
2229.5 1885.9 841.8 1962.7 1832.4 5555.2 4265.7 1611.4 1888.1 0.0 740.9 1097.3 1893.1 2120.5 2104.9 1332.2 1109.9 202.6 339.4 900.1
2533.6 2569.1 402.7 2687.1 2103.2 6090.6 4869.9 2208.4 2572.5 740.9 0.0 1426.0 2326.7 2698.3 2820.8 2032.1 1736.5 538.6 480.1 826.7
3248.2 2429.7 1148.6 1804.7 2881.1 6238.4 4856.3 2391.5 2427.8 1097.3 1426.0 0.0 904.9 1377.2 2483.0 1910.8 727.3 1140.7 1029.3 712.1
3819.7 2632.4 2050.6 1562.7 3508.8 6332.1 4909.4 2788.6 2627.5 1893.1 2326.7 904.9 0.0 662.2 2544.7 2231.5 946.5 1985.7 1904.8 1599.6
3713.3 2309.2 2494.5 1060.5 3465.2 5842.5 4417.9 2593.1 2302.6 2120.5 2698.3 1377.2 662.2 0.0 2135.7 2027.5 1014.0 2262.8 2232.5 2085.5
2025.6 387.8 2934.7 1091.0 2001.7 3801.8 2391.3 956.9 379.0 2104.9 2820.8 2483.0 2544.7 2135.7 0.0 795.2 1785.4 2301.7 2436.5 2832.6
1699.9 559.2 2172.1 1215.3 1515.3 4339.7 2987.4 567.5 560.5 1332.2 2032.1 1910.8 2231.5 2027.5 795.2 0.0 1319.5 1523.2 1669.7 2126.5
 2436.5 1669.7 1258.1 185.3 0.0 626.2
3109.8 2684.0 468.2 2371.4 2698.9 6430.8 5109.5 2487.7 2684.6 900.1 826.7 712.1 1599.6 2085.5 2832.6 2126.5 1286.6 807.2 626.2 0.0

Rute Pelayaran Optimal:
Kota Omashu --> Desa Tu Zin Cebu --> Kota Davao Kerajaan Bumi --> Kota Puerto Princesa Palawan
 --> Kota Chin Manila --> Desa Senlin --> Kota Ba Sing Se --> Kota Saigon --> Kota Phnom Penh
Kerajaan Bumi --> Desa Full Moon Bay Kerajaan Bumi --> Kota Kyoshi Bangkok --> Desa Pulau Kyos
hi --> Desa Pulau Ember Langkawi --> Kota Republik Singapore --> Kota Kuching Borneo --> Kota
Jakarta Kerajaan Bumi --> Kota Surabaya Kerajaan Bumi --> Kota Gaoling --> Kota Omashu

Jarak Total Rute Pelayaran: 12639.032 km


Kota yang Tidak Dilewati:
1. Desa Makapu Oahu
2. Desa Hira'a Tahiti
```

4.4.2. Waktu Eksekusi

Waktu eksekusi program diukur dengan menggunakan fungsi clock() pada library time.h dan pengukuran dilakukan pada visual studio code versi 1.77.33. Pengukuran dilakukan sebanyak 10 kali untuk mengambil rerata waktu eksekusi dari program

Pengukuran ke-	*Waktu Eksekusi (s)*
1	**0,039**
2	**0,09**
3	**0,047**
4	**0,158**
5	**0,163**
6	**0,039**
7	**0,098**
8	**0,035**
9	**0,032**
10	**0,107**

Dari 10 pengukuran didapatkan rata-rata waktu yang dibutuhkan untuk eksekusi program adalah 0,08 detik.

5. KESIMPULAN

- Dalam memecahkan masalah kompleks seperti Travelling Salesman Problem, dilakukan langkah pengerjaan yang terdiri dari mendefinisikan ruang lingkup masalah, merancang software, implementasi rancangan, dan pengujian.
- Permasalahan Travelling Salesman Problem dapat diselesaikan menggunakan algoritma Nearest Neighbor yang mencari jarak terpendek dari setiap pelabuhan ke pelabuhan selanjutnya.
- Proses penyelesaian dapat dibagi menjadi 7 proses yang terdiri dari memindahkan isi file ke dalam array (filetoArray), menghitung jarak dari titik-titik koordinat pelabuhan (hitungJarak), membuat matriks jarak dari data pelabuhan pada array (arraytoMatrix), menentukan pelabuhan yang menjadi titik awal (harbor_check), menentukan pelabuhan selanjutnya (find_next_h), menentukan solusi rute pelayaran (tsp), dan menentukan pelabuhan yang tidak dilewati (skipped_harbor).

DAFTAR PUSTAKA

[1] Hidayat, A., Hartati, S., & Sari, R. M. (2019). Penerapan algoritma ant colony system (ACS) untuk menyelesaikan travelling salesman problem (TSP). Semantik, 5(2), 87-98. https://doi.org/10.32696/semantik.v5i2.2617

[2] Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., dan Shmoys, D. B. (1985). The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. John Wiley & Sons, Inc.

[3] A. Fitria, Muslim, and H. Azis, "Analisis Kinerja Sistem Klasifikasi Skripsi menggunakan Metode Naïve Bayes Classifier," vol. 3, no. 2, pp. 102–106, 2018.

[4] Fetty Fitriyanti Lubis. 26 October 2022. Struktur Diskrit. Institut Teknologi Bandung, Kota Bandung.

LAMPIRAN

Link Repository git:

https://github.com/el2208-ppmc-2023/modul-9-team-b5

File pelabuhan.csv

```
Pelabuhan,Lintang,Bujur
Kota Ba Sing Se,25.0330,121.5654
Kota Omashu,10.3157,123.8854
Desa Pulau Kyoshi,10.4744,98.9305
Kota Gaoling,-0.9116,119.9004
Desa Senlin,23.5565,117.6227
Desa Makapu Oahu,21.3394,157.7147
Desa Hira'a Tahiti,15.2549,145.8150
Kota Chin Manila,14.5995,120.9842
Desa Tu Zin Cebu,10.2352,123.901
Kota Saigon,10.8231,106.6297
Kota Kyoshi Bangkok,13.7563,100.5018
Kota Republik Singapore,1.3521,103.8198
Kota Jakarta Kerajaan Bumi,-6.2088,106.8456
Kota Surabaya Kerajaan Bumi,-7.2575,112.7521
Kota Davao Kerajaan Bumi,7.1907,125.4553
Kota Puerto Princesa Palawan,9.9672,118.7859
Kota Kuching Borneo,1.5497,110.3631
Kota Phnom Penh Kerajaan Bumi,11.5564,104.9282
Desa Full Moon Bay Kerajaan
Bumi,10.6093,103.5297
Desa Pulau Ember Langkawi,6.3500,99.8000
```