Aplikasi Pohon Terurut untuk Meningkatkan Efisiensi Algoritma Djikstra

David Karel Halomoan - 13520154
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13520154@std.stei.itb.ac.id

***Abstract*—Banyak permasalahan di dunia ini yang dapat diselesaikan dengan komputer seiring berkembangnya ilmu komputer dan kekuatan komputasi. Salah satu algoritma yang banyak digunakan untuk menyelsaikan berbagai permasalahn ini adalah algoritma Djikstra. Algoritma Djikstra adalah algoritma yang digunakan untuk mencari panjang lintasan terpendek dari suatu simpul ke simpul lain dalam suatu graf berbobot. Pada makalah ini akan dibahas mengenai pemanfaatan phon terurut untuk meningkatkan efisiensi algoritma Djikstra secara drastis. Penulis juga menyediakan *link repository*implementasi kode dalam Bahasa C++.**

***Keywords*—Djikstra, graf, peta, C++.**

I. PENDAHULUAN

Perkembangan ilmu komputer dan kekuatan komputer telah memudahkan kita memecahkan berbagai masalah. Salah satu masalah yang dapat diselesaikan dengan komputer adalah pencarian rute tedekat dari suatu tempat ke tempat lainnya. Salah satu algoritma yang dapat menyelesaikan masalah ini adalah algoritma Djikstra.

Algoritma Djikstra dilakukan dengan menggambarkan peta sebagai graf berarah yang berbobot. Algoritma Djikstra memiliki berbagai macam implementasi. Salah satu implementasi yang banyak digunakan pada algoritma Djikstar adalah dengan menggunakan *priority queue*. Pada makalah ini, penulis akan membahas pengimplementasian pohon terurut sebagai *priority queue* pada algoritma Djikstra.

Penulis juga akan membahas mengenai kompleksitas algoritma. Pembahasan ini dilakukan karena akan dilakukan analisis kompleksitas algoritma terhadap pengimplementasian pohon terurut sebagai *priority queue* pada algoritma Djikstra untuk dibandingkan dengan salah satu pengimplementasian lain.

II. LANDASAN TEORI

A. Graf

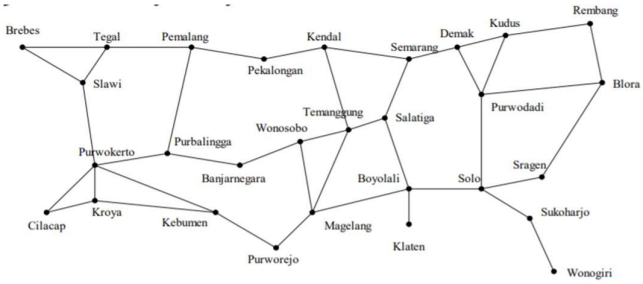
1. Definisi Graf

Graf banyak digunakan untuk menggambarkan objek-objek diskrit (tidak saling berhubungan, lawan dari kontinu) dan hubungan antara objek-objek tersebut.

Graf terdiri dari simpul (*vertex*) dan sisi (*edge*). Graf G dapat didefinisikan sebagai *tuple* (V,E), yang dalam hal ini:

V = himpunan tidak kosong dari simpul-simpul = $\{V_1, V_2, \dots, V_n\}$

G = himpunan sisi yang menghubungkan sepasang simpul = $\{e_1, e_2, \dots, e_n\}$

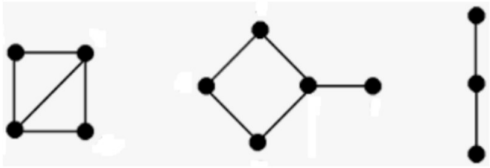


Gambar 2.1 Graf yang menunjukkan peta jaringan jalan raya sejumlah kota di Provinsi Jawa Tengah
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

2. Jenis-jenis Graf

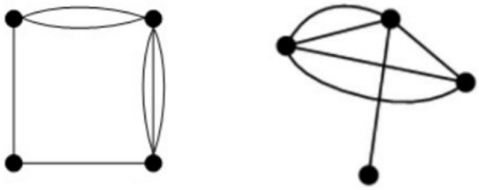
Berdasarkan ada atau tidaknya gelang atau sisi ganda pada suatu graf, grak digolongkan menjadi dua jenis:

- Graf sederhana
  Graf yang tidak mengandung gelang maupun sisi ganda dinamakan graf sederhana.



Gambar 2.2 Contoh graf sederhana
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

- Graf tak-sederhana
  Graf yang mengandung sisi ganda atau gelang dinamakan graf tak-sederhana.



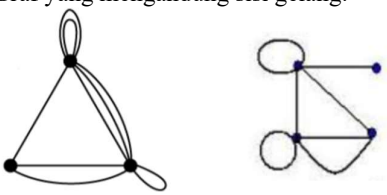
Gambar 2.3 Contoh graf tak-sederhana
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

Graf tak-sederhana dibedakan lagi menjadi:

- Graf ganda (*multi-graph*)
  Graf yang mengandung sisi ganda.

Gambar 2.3 Contoh graf ganda
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-

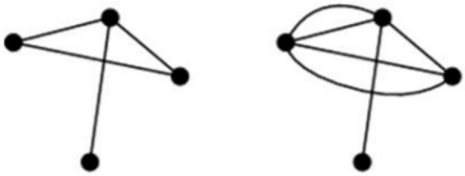
- Graf semu (*pseudo-graph*)
  Graf yang mengandung sisi gelang.



Gambar 2.4 Contoh graf semu
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-

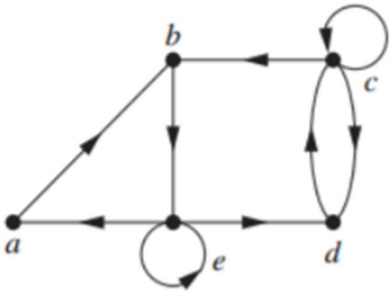
Berdasarkan orientasi arah pada sisi, graf dibedakan atas 2 jenis:

- Graf tak-berarah
  Graf yang sisinya tidak mempunyai orientasi arah disebut graf tak-berarah.



Gambar 2.5 Contoh graf tak-berarah
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

- Graf berarah
  Graf yang setiap sisinya diberikan orientasi arah disebut graf berarah.



Gambar 2.5 Contoh graf berarah
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

3. Terminologi Graf

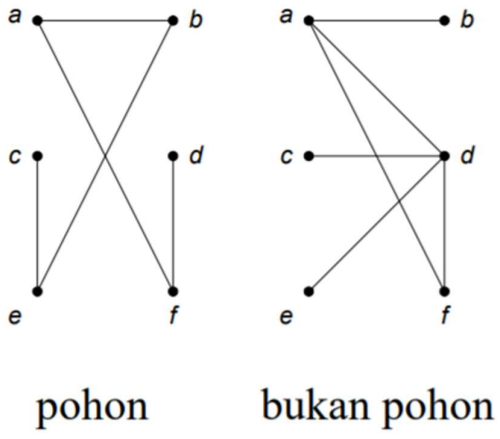
Beberapa terminologi dalam graf:

- Ketetanggan (*adjacent*)
  Dua buah simpul dikatakan bertetangga bila keduanya terhubung langsung oleh suatu sisi.
- Bersisian (*adjacency*)
  Sebuah sisi dikatakan bersisian dengan suatu simpul bila sisi tersebut berhubungan langsung dengan simpul tersebut (terdapat di ujung-ujung sisi).
- Simpul Terpencil (*Isolated Vertex*)
  Simpul terpencil adalah simpul yang tidak mempunyai sisi yang bersisian dengannya.
- Graf kosong (*null/empty graph*)
  Graf yang himpunan sisinya merupakan himpunan kosong (tidak memiliki sisi).
- Derajat (*degree*)
  Derajat suatu simpul adalah jumlah sisi yang bersisian dengan simpul tersebut.
- Lintasan (*Path*)
  Lintasan dari simpul awal v_0 ke simpul tujuan v_n di dalam graf G dengan panjang n adalah barisan berselang-seling simpul-simpul dan sisi-sisi yang berbentuk
  $$v_0, e_1, v_1, e_2, v_2, \dots, v_{n-1}, e_n, v_n$$
  dengan
  $e_1 = (v_1, v_2), e_2 = (v_1, v_2), \dots, e_n = (v_{n-1}, v_n)$
  adalah sisi-sisi dari graf G.
- Siklus atau Sirkuit
  Lintasan yang berawal dan berakhir pada simpul yang sama disebut siklus atau sirkuit.
- Keterhubungan
  Dua buah simpul v_x dan v_y disebut terhubung jika terdapat lintasan dari v_x ke v_y.

B. Pohon

1. Definisi Pohon

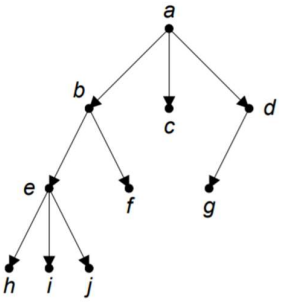
Pohon adalah graf tak-berarah terhubung yang tidak mengandung sirkuit.



Gambar 2.6 Contoh graf pohon dan bukan pohon
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf

2. Pohon Berakar

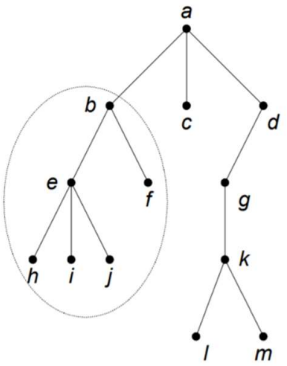
Pohon yang satu buah simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi akar sehingga menjadi graf berarah dinamakan pohon berskar. Sebagai perjanjian, tanda panah pada sisi dapat dibuang. Pohon yang akan dibahas dalam makalah ini adalah pohon berakar.



Gambar 2.7 Contoh graf pohon berakar
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf

Beberapa terminologi dalam pohon berakar:

- Anak dan Orangtua
  Contohnya pada gambar 2.7, simpul b, c, dan d adalah anak-anak simpul a dan a adalah orangtua dari anak-anak itu.
- Lintasan
  Contohnya pada gambar 2.7, lintasan dari a ke j adalah a, b, e, j. Panjang lintasan dari a ke j adalah 3.
- Saudara Kandung
  Contohnya pada gambar 2.7, f adalah saudara kandung e, tetapi g bukan saudara kandung e, karena orangtua mereka berbeda.
- Upapohon.
  Upapohon adalah bagian dari suatu pohon yang berbentuk pohon juga.



Gambar 2.8 Upapohon pada suatu pohon
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf

- Derajat
  Derajat sebuah simpul adalah jumlah upapohon (anak) pada simpul tersebut.
- Daun
  Daun adalah simpul yang berderajat nol (tidak mempunyai anak).
- Simpul Dalam
  Simpul dalam adalah simpul yang mempunyai anak.
- Aras atau tingkat
  Contoh pada gambar 2.7, simpul a memiliki tingkat 0, simpul, b, c, dan d memiliki tingkat 1, dan seterusnya.
- Tinggi atau kedalaman
  Tinggi adalah tingkat maksimum dari suatu pohon. Pohon pada gambar 2.7 memiliki tinggi 4.

3. Pohon Terurut

Pohon terurut adalah pohon yang urutan anak-anaknya penting (tidak bisa sembarangan).

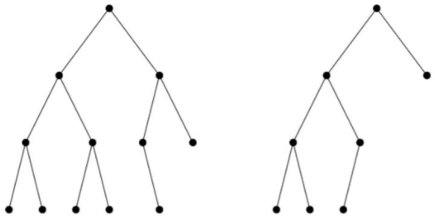
4. Pohon *n*-ary

Pohon berakar yang setiap simpulnya paling banyak memiliki 2 anak disebut pohon *n*-ary.

5. Pohon Biner (*binary*)

Pohon biner adalah pohon yang setiap simpulnya maksimal memiliki 2 anak. Pohon biner adalah pohon *n*-ary dengan n = 2. Anak pada pohon biner dibedakan dengan istilah anak kanan dan anak kiri. Perbedaan urutan anak ini menyebabkan pohon biner adalah sebuah pohon terurut.

Pada pohon biner ada istilah pohon biner seimbang, yaitu pohon yang memiliki perbedaan tinggi upapohon kiri dan upapohon kanan maksimal 1.



Gambar 2.9 Pohon sebelah kiri adalah pohon seimbang sedangkan pohon di sebelah kanan tidak seimbang
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf

C. Kompleksitas Algoritma

Kompleksitas algoritma digunakan untuk mengukur efisiensi suatu algoritma. Kebutuhan waktu dari suatu algoritma bergantung kepada banyaknya data masukan yang diproses oleh algoritma, ukuran ini dinotasikan dengan n. Kompleksitas algoritma diperlukan untukmemilih algoritma terbaik dari berbagai algoritma yang dapat memecahkan suatu permasalahan.

Kompleksitas waktu, $T(n)$, diukur sebagai jumlah tahapan komputasi yang diperlukan suatu algoritma untuk ukuran masukan n. Untuk menyederhanakan perhitungan, yang dihitung hanya jumlah operasi yang mendasari algoritma itu saja, tidak semua operasi.

Selain kompleksitas waktu biasa, juga perlu mengetahui kebutuhan waktu algoritma Ketika ukuran masukannya (n) naik. Kinerja suatu algoritma biasanya baru akan tampah untuk n yang sangat besar, bukan pada n kecil. Notasi kompleksitas waktu algoritma untuk n yang sangat besar dinamkan kompleksitas waktu asimptotik.

Notasi "*O-Besar*" (*Big-O*) merupakan notasi kompleksitas waktu asimptotik. Definisi notasi Big-O:

$$T(n) = O(f(n)) \; bila \; terdapat \; konstanta \; C \; dan \; n_0 \; sedemikian \; sehingga \; T(n) \leq Cf(n) \; untuk \; n \geq n_0.$$

Dari sini bisa disimpulkan bahwa $f(n)$ adalah batas lebih atas (*upper bound*) dari $T(n)$ untuk n yang besar.

D. Algoritma Djikstra

Banyak algoritma yang dapat menyelesaikan permasalahan lintasan terpendek. Salah satu algoritma tersebut adalah algoritma Djikstra. Algoritma ini ditemukan oleh Edsger Djikstra, seorang ilmuwan komputer asala Belanda, pada tahun 1959. Pada makalah ini algoritma Djikstra akan digunakan untuk mencari lintasan terpendek pada suatu graf berarah yang berbobot.

Algoritma Djikstra tidak hanya mencari lintasan terpendek dari simpul awal ke simpul akhir yang sudat ditentukan, tetapi juga lintasan terpendek dari simpul tujuan ke semua simpul yang ada pada graf.

Diperlukan *hash table* pada algoritma ini, yaitu struktur data yang merupakan pasangan *key-value*. Terdapat dua *hash table* yang akan digunakan, yaitu *shortest_path* dan *shortest_previous_stopover*, *shortest_path* akan berisi daftar jarak terpendek dari simpul awal ke simpul tujuan, *key* pada *shortest_path* berisi nama simpul tujuan dari simpul awal, sedangkan *value* pada *shortest_path* berisi panjang lintasan terpendek dari simpul awal ke simpul *key* dari *value* tersebut, sedangkan *shortest_previous_stopover* akan berisi daftar simpul yang harus dilewati jika ingin menuju simpul tujuan dengan lintasan terpendek, *key* pada *shortest_previous_stopover* berisi nama simpul tujuan yang akan dituju dari simpul awal, sedangkan *value* pada *shortest_previous_stopover* adalah simpul yang harus dilalui dari simpul awal jika ingin mencapai simpul pada *key* dengan lintasan terpendek.

Berikut tahapan algoritma Djikstra:

1. Simpul pertama dikunjungi, dijadikan sebagai *current_vertex* (simpul yang sedang dikunjungi).
2. Panjang lintasan dari *current_vertex* ke tiap simpul tetangga dicek.
3. Jika panjang lintasan dari simpul awal ke sebuah simpul yang bertetangga dengan *current_vertex* lebih pendek atau jika simpul tersebut tidak ada pada *shortest_path*,
   a. Lakukan *update* pada *shortest_path* sesuai dengan lintasan terpendek.
   b. Lakukan *update* pada *shortest_previous_stopover* dengan menjadikan simpul tetangga sebagai *key* dan *current_vertex* sebagai *value*.
4. Lalu, simpul dengan lintasan terpendek dari simpul awal yang belum pernah dikunjungi dan dijadikan *current_vertex*.
5. Ulangi langkah 2 sampai 4 sampai semua simpul telah dikunjungi.

III. DESKRIPSI MASALAH

Pada algoritma Djikstra, setiap kali kita ingin berpindah ke simpul baru untuk dijadikan *current_vertex*, kita harus melakukan iterasi pada sisa simpul yang belum dikunjungi dan menentukan mana yang memiliki lintasan terpendek dari simpul awal. Jika diimplementasikan menggunakan *array*, tentu harus diperhitungkan juga operasi pergeseran saat dilakukan penghapusan elemen dari sisa simpul yang belum dikunjungi. Ini mungkin tidak terlalu berpengaruh untuk graf yang tidak terlalu besar, tetapi ini dapat mempengaruhi efisiensi algoritma untuk graf besar.

Untuk mengatasi hal ini, akan digunakan *priority queue*, yaitu struktur data yang mirip dengan *queue* yang memiliki prinsip FIFO (*First In First Out*), hanya saja setiap kali data dimasukkan data tersebut akan diurutkan di dalam *priority queue* sesuai aturan yang ditentukan. Untuk implementasi yang akan dibahas, data akan diurutkan menurut panjang lintasan suatu simpul dari simpul awal, dengan panjang lintasan terkecil berada di "depan" *priority queue* dan terurut membesar sampai ke "belakang". Pengimplementasian seperti ini menyebabkan algoritma hanya harus mengecek elemen yang berada di "depan" *priority queue* saja karena sudah dipastikan elemen tersebut adalah simpul dengan panjang lintasan terkecil dari simpul awal yang belum dikunjungi. Pada makalah ini, *priority*

queue akan diimplementasikan dengan sebuah pohon terurut yang disebuat *heap*, implementasi dengan *heap* memiliki berbagai keunggulan dibandingkan implementasi dengan *array* biasa.

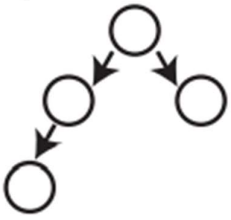
IV. Analisis dan Implementasi

A. Heap

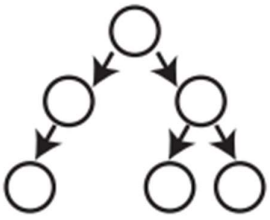
Heap yang akan dibahas di makalah ini adalah *heap* biner (*binary heap*). *Heap* biner adalah sebuah pohon biner, hanya saja dengan aturan khusus yaitu:

- Nilai pada tiap simpul harus lebih besar dari nilai tiap keturunan dari simpul tersebut.
- Pohon harus *lengkap*.

Pohon lengkap adalah pohon yang *dipenuhi* oleh simpul. Pohon lengkap di sini dapat memiliki posisi-posisi simpul yang kosong dengan syarat tidak ada simpul lain di sebalah *kanan* posisi tersebut.



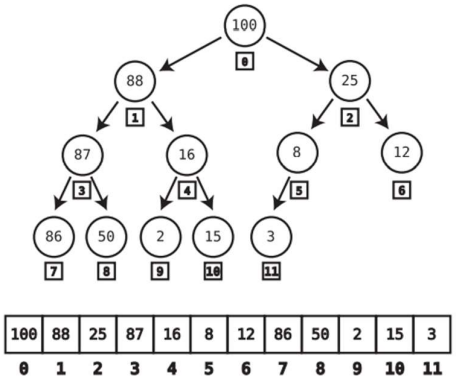
Gambar 4.1 Contoh pohon biner lengkap
Sumber:
A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Programming Skills, halaman 283



Gambar 4.2 Contoh pohon biner tidak lengkap
Sumber:
A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Programming Skills, halaman 283

Heap diimplementasikan dengan *array* dengan elemen pada *array* terurut berdasarkan tingkat pada pohon biner. Elemen pertama pada *array* berisi elemen akar (*root*) dari pohon biner. Elemen berikutnya adalah simpul pada tingkat berikutnya yang diurutkan dari *kiri*.

Jika dilihat pada gambar 4.3, jika kita melakukan iterasi terhadap elemen dengan *array* dengan indeks 0 sampai indeks akhir, itu akan sama saja dengan kita melakukan BFS (*Breadth First Search*) pada pohon biner. Simpul yang berada pada indek terakhir disebut dengan simpul terakhir (*last node*).



Gambar 4.3 Implementasi array sebagai pohon biner
Sumber:
A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Programming Skills, halaman 296

Untuk mencari anak kiri dari suatu simpul yang berada pada indeks i, digunakan rumus:

$$indeks\ anak\ kiri = (i * 2) + 1$$

, sedangkan untuk mencari anak kanan digunakan rumus:

$$indeks\ anak\ kanan = (i * 2) + 2$$

, untuk mencari orang tua digunakan rumus:

$$indeks\ orang\ tua = \frac{(i-1)}{2}$$

, rumus indeks orang tua menggunakan pembagian integer (bilangan bulat), yaitu dilakukan pembulatan ke bawah saat didapatkan hasil bukan bilangan bulat. Simbol $*$ melambangkan perkalian.

Terdapat dua operasi utama pada *heap* yatu *insertion* (memasukkan simpul baru pada *heap*) dan *deletion* (menghapus simpul akar dari *heap*).

Untuk melakukan *insertion*, tahapannya adalah:

1. Simpul dengan nilai yang baru dibuat dan dimasukkan pada posisi kosong paling kiri pada tingkat bawah *heap* (menambahkan elemen baru di akhir *array*).
2. Bandingkan simpul baru dengan simpul orangtuanya.
3. Jika nilai pada simpul baru lebih kecil dari nilai pada simpul orangtuanya, tukar simpul baru dengan simpul orangtuanya.
4. Ulangi langkah 3 sampai didapatkan simpul orang tua dengan nilai yang lebih kecil atau sama dengan nilai simpul baru atau simpul baru berada di akar (simpul baru *naik* pada *heap*).

Insertion pada *heap* memiliki kompleksitas waktu $O(\log n)$. Operasi khas dalam algoritma ini adalah pertukaran simpul baru dengan simpul orangtuanya. Jika sebuah *heap* memiliki n buah simpul, simpul-simpul tersebut akan tersusun ke dalam $\log n$ tingkat sehingga algoritma pada *worst-case scenario* melakukan $\log n$ operasi pertukaran.

Deletion pada *heap* hanya dapat dilakukan pada simpul akarnya saja. Untuk melakukan *deletion*, tahapannya adalah:

1. Pindahkan simpul terakhir (elemen terakhir pada *array*) ke akar, dengan ini akar terhapus, simpul ini akan disebut dengan simpul alir.
2. *Alirkan* simpul alir ke bawah sampai posisi yang seharusnya. *Pengaliran* dilakukan dengan cara:
    1. Periksa kedua anak simpul alir untuk menentukan anak dengan nilai yang lebih besar.
    2. Jika simpul alir lebih kecil dari pada anak dengan nilai lebih besar yang ditentukan pada tahap 1, tukar simpul alir dengan simpul anak tersebut.
    3. Ulangi langkah 1 dan 2 hingga simpul alir tidak memiliki anak dengan nilai yang lebih besar dari nilai simpul alir tersebut.

Deletion pada *h*eap memiliki kompleksitas waktu $O(\log n)$. Operasi khas dalam algoritma ini adalah pertukaran simpul alir dengan salah satu simpul anaknya. Jika sebuah *heap* memiliki n buah simpul, simpul-simpul tersebut akan tersusun ke dalam $\log n$ tingkat sehingga algoritma pada *worst-case scenario* melakukan $\log n$ operasi pertukaran.

Jika dibandingkan dengan *ordered array* (*array* terurut) yang melakukan *deletion* pada akhir *array*, hasilnya akan seperti ini:

	Ordered Array	*Heap*
Insertion	$O(n)$	$O(\log n)$
Deletion	$O(1)$	$O(\log n)$

Jika dilihat sekilas, perbedaannya mungkin tidak terlalu mencolok. *Ordered array* lebih cepat dari pada *heap* dalam operasi *deletion*, *heap* lebih cepat dari pada *ordered array* dalam operasi *insertion*. Hanya saja, yang harus diperhitungkan adalah $O(\log n)$ jauh lebih cepat dari pada $O(n)$ untuk n yang bernilai besar. Untuk $n = 1000$, $\log 1000 \approx 9.966$ (ingat log di sini memiliki basis 2), sedangkan untuk $n = 1000000$, $\log 1000000 \approx 19.932$. Ini tentu jauh lebih cepat dibandingkan dengan $O(n)$ sehingga operasi *insertion* pada *heap* jauh lebih cepat dibandingkan *ordered array* yang menyebabkan perbedaan efisiensi *deletion* yang tidak begitu besar dapat diabaikan, pertimbangan inilah yang membuat *heap* lebih baik dibandingkan *ordered array*.

Heap memiliki konsistensi operasi yang cepat dibandingkan dengan *ordered array* yang dapat menjadi lambat saat n bernilai besar.

B. Implementasi Pada Bahasa C++

Kode program terdapat pada *repository* *https://github.com/davidkarelh/Heap-Implementation-for-Djikstra-Algorithm*.

Simpul pada pohon diimplementasikan dengan membuat kelas Bernama Vertex, berikut definisi kelas Vertex:

```
class Vertex {
    public:
        string name;
        map<Vertex *, int> routes;

        Vertex() {};

        Vertex(string vertex_name) {
            name = vertex_name;
        }

        void add_route(Vertex * vertex_pointer, int path_length) {
            routes[vertex_pointer] = path_length;
        }
};
```

Gambar 4.4 Implementasi kelas Vertex pada C++

Sumber: Komputer penulis

Vertex di sini memiliki nama simpul dan *routes* yang berisi pointer terhadap Vertex lain dan bobot lintasan dari Vertex ini ke Vertex lain tersebut.

Graf diimplementasikan dengan representasi *adjacency list,* setiap simpul memiliki trailing list (diimplementasikan dengan *map* Bernama *routes* pada kelas Vertex) yang berisi *pointer* keVertex lain. Graf diimplementasikan dengan kelas Graph, berikut definisi kelas Graph:

```
class Graph {
    public:
        Vertex adjacency_list[50];
        int length;

        Graph() {
            length = 0;
        };

        void add_vertex(Vertex v) {
            adjacency_list[length] = v;
            length++;
        }

        // Doesn't fully delete a vertex from Graph, only delete
        // the specified vertex from the adjacency list,
        // deleted vertex may still exist in the remaining vertices routes
        void delete_vertex(string name) {
            int i, j;
            for (i = 0; i < length ; i++) {
                if (adjacency_list[i].name == name) {
                    for (j = i; j < length - 1; j++) {
                        adjacency_list[j + 1] = adjacency_list[j];
                    }
                    break;
                }
            }
        }
};
```

Gambar 4.5 Implementasi kelas Graph pada C++

Sumber: Komputer penulis

Heap diimplementasikan berisi elemen bertipe *ElType* dengan definisi sebagai berikut:

```
typedef struct {
    string from;
    Vertex * to;
    int path_length;
} ElType;
```

Gambar 4.6 Definisi ElType
Sumber: Komputer penulis

from berisi nama simpul yang dilewati, *to* berisi *pointer* ke Vertex (simpul) tujuan, dan *path_length*

menyatakan panjang lintasan dari simpul awal (bukan simpul yang dilewati) ke simpul tujuan. Uritan pada *priority queue* disesuaikan dengan *path_length* pada setiap elemen.

Berikut properti pada kelas Heap:

```
class Heap {

    public:
        ElType data[50];
        int length;
```

Gambar 4.7 Properti pada kelas Heap

Sumber: Komputer penulis

Data merupakan *array* tempat disimpannya elemen *heap* sesuai gambar 4.3, sedangkan *length* berisi panjang *array* yang digunakan.

Heap memiliki *method* insert_data yang berguna untuk memasukkan *ElType* bar uke dalam *heap* sesuai aturan, berikut *method* tersebut:

```
void insert_data(ElType new_data) {
    int new_node_index;
    ElType temp;

    data[length] = new_data;
    new_node_index = length;
    length++;

    while (new_node_index > 0 && data[new_node_index].path_length < data[parent_idx(new_node_index)].path_length) {
        temp = data[parent_idx(new_node_index)];
        data[parent_idx(new_node_index)] = data[new_node_index];
        data[new_node_index] = temp;

        new_node_index = parent_idx(new_node_index);
    }
}
```

Gambar 4.8 Method insert_data pada kelas Heap

Sumber: Komputer penulis

*H*eap juga memiliki *method* delete_data yang menghapus *ElType* yang berada pada *root* sesuai aturan dan mengembalikannya. berikut *method* tersebut:

```
ElType delete_data() {
    ElType temp, val;
    int trickle_node_idx, lesser_child_idx;

    val = root_node();
    data[0] = last_node();
    length--;

    trickle_node_idx = 0;

    while (has_lesser_child(trickle_node_idx)) {
        lesser_child_idx = calculate_lesser_child_idx(trickle_node_idx);

        temp = data[trickle_node_idx];
        data[trickle_node_idx] = data[lesser_child_idx];
        data[lesser_child_idx] = temp;

        trickle_node_idx = lesser_child_idx;
    }
    return val;
}
```

Gambar 4.9 Method delete_data pada kelas Heap
Sumber: Komputer penulis

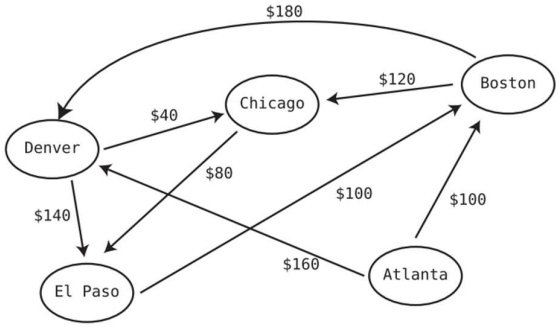
Priority queue diimplementasikan dengan kelas PrioQueue, berikut implementasinya:

```
class PrioQueue {
    Heap pqueue;

    public:
        bool isEmpty() {
            return pqueue.length == 0;
        }

        void enqueue(ElType val) {
            pqueue.insert_data(val);
        }

        ElType dequeue() {
            return pqueue.delete_data();
        }
};
```

Gambar 4.10 Implementasi kelas PrioQueue pada C++
Sumber: Komputer penulis

Pada program terdapat dua fungsi dengan algortima Djikstra, yaitu *djikstra_shortest_path* dan *djikstra_shortest_path2*. Fungsi *djikstra_shortest_path* diimplementasikan dengan *array*, sedangkan fungsi *djikstra_shortest_path2* diimplementasikan dengan *heap*.

Dilakukan pengecekan program terhadap sebuah graf *test case*, yaitu:



Gambar 4.11 Graf test case
Sumber:
A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Programming Skills, halaman 367

Program akan mencari lintasan terpendek dari Atlanta ke El Paso beserta rutenya.

Hasil *djikstra_shortest_path* sebagai berikut:

```
PS D:\Makalah Matdis> g++ main.cpp -o main; .\main
Shortest path from Atlanta to:
Atlanta 0
Boston 100
Chicago 200
Denver 160
El Paso 280

Shortest previous stopover vertex from Atlanta:
To Boston from Atlanta
To Chicago from Denver
To Denver from Atlanta
To El Paso from Chicago

Shortest path from Atlanta to El Paso: Atlanta -> Denver -> Chicago -> El Paso
Shortest path weight from Atlanta to El Paso: 280
```

Gambar 4.12 Hasil implementasi algoritma Djikstra dengan array
Sumber: Komputer penulis

Hasil dari *djikstra_shortest_path2* sebagai berikut:

```
PS D:\Makalah Matdis> g++ main.cpp -o main; .\main
Shortest path from Atlanta to:
Atlanta 0
Boston 100
Chicago 200
Denver 160
El Paso 280

Shortest previous stopover vertex from Atlanta:
To Atlanta from Atlanta
To Boston from Atlanta
To Chicago from Denver
To Denver from Atlanta
To El Paso from Chicago

Shortest path from Atlanta to El Paso: Atlanta -> Denver -> Chicago -> El Paso
Shortest path weight from Atlanta to El Paso: 280
```

Gambar 4.13 Hasil implementasi algoritma Djikstra dengan heap
Sumber: Komputer penulis

Kedua program menghasilkan hasil yang sama dan benar, terbukti bahwa algoritma Djikstra dapat diimplementasikan dengan *heap* tanpa mengurangi ketepatan algoritma.

V. KESIMPULAN DAN SARAN

Algoritma Djikstra merupakan salah satu algoritma yang dapat membantu memecahkan berbagai macam permasalahan. Dari analisis ditemukan efisiensi algoritma Djikstra dapat ditingkatkan secara drastis dengan menimplementasikan pohon berurut sebagai *priority queue* pada algoritma Djikstra. Algoritma Djikstra dengan implementasi pohon terurut sebagai *priority queue* juga telah berhasil diimplementasikan ke dalam Bahasa C++ tanpa mengurangi ketepatan algoritma.

Untuk penelitian lebih lanjut, jumlah langkah algoritma dengan implementasi *array* dan implementasi *heap* dapat ditunjukkan untuk melihat perbedaan efisiensi algoritma.

VI. UCAPAN TERIMA KASIH

Penulis mengucapkan puji syukur kepada Tuhan Yang Maha Esa, karena atas kasih, rahmat, dan karunia-Nya sehingga penulis dapat menyelesaikan makalah ini. Penulis juga mengucapkan terima kasih kepada Ibu Dr. Nur Ulfa Maliadevi, S.T, M.Sc, dosen kelas 3 mata kuliah Matematika Diskrit IF2120 atas bimbingan dan pengajarannya selama ini. Penulis juga mengucapkan terima kasih kepada para penulis referensi makalah ini karena tanpa karya mereka, makalah ini tidak akan jadi.

REFERENSI

[1] Wengrow, Jay. 2020. *A Common-Sense Guide to Data Structures and Algorithms: Level Up Your Core Programming Skills, Second Edition*. Pragmatic Bookshelf.

[2] Munir, Rinaldi. 2021. Graf(Bagian 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf (diakses pada 13 Desember 2021)

[3] Munir, Rinaldi. 2021. Pohon(Bagian 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf (diakses pada 13 Desember 2021)

[4] Munir, Rinaldi. 2021. Pohon(Bagian 2). https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf (diakses pada 13 Desember 2021)

[5] Munir, Rinaldi. 2021. Pohon(Bagian 2). https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf (diakses pada 13 Desember 2021)

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.

Bogor, 14 Desember 2021

David Karel Halomoan
13520154