Tugas Mandiri

Perancangan Dan Analisis Algotirma

Dosen Pengampu:

Randi Proska Sandra, S.Pd., M.Sc.

Oleh:

Maharani Safitri

21343008

INFORMATIKA

JURUSAN TEKNIK ELEKTRONIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2023

A. Penjelasan Algoritma Dijkstra

Algoritma Dijkstra adalah algoritma yang digunakan untuk mencari jalur terpendek dari suatu node ke semua node lain dalam sebuah graf berarah dengan bobot yang tidak negatif. Dalam algoritma ini, bobot setiap sisi merepresentasikan jarak antara dua node pada graf. Konsep dasar dari algoritma Dijkstra meliputi definisi graf, sisi (edge), node (vertex), bobot pada setiap sisi, dan jarak terpendek antar node. Graf adalah kumpulan node dan sisi yang menghubungkan antara satu node dengan node yang lain. Sisi menghubungkan dua node pada graf dan bobot pada setiap sisi adalah nilai numerik yang menunjukkan jarak atau biaya yang diperlukan untuk bergerak dari satu node ke node lainnya. Node adalah titik pada graf yang merepresentasikan suatu entitas seperti kota atau bangunan. Jarak terpendek antar node adalah jarak atau biaya terkecil yang harus ditempuh untuk mencapai node tujuan dari node awal.

Langkah pertama dalam algoritma Dijkstra adalah menentukan node awal dan memberikan nilai nol pada node tersebut. Langkah kedua adalah menentukan jarak terpendek dari node awal ke setiap node yang berdekatan dengan node tersebut. Setelah itu, nilai jarak terpendek tersebut akan di-update pada node terkait. Langkah berikutnya adalah memilih node dengan jarak terpendek yang paling rendah dan memasukkannya ke dalam daftar node yang telah diproses. Kemudian, nilai jarak terpendek pada node-node tetangga akan diperbarui dengan mengambil nilai jarak terpendek dari node terpilih dan menambahkannya dengan nilai bobot sisi antara node terpilih dan node tetangga tersebut. Langkah-langkah tersebut diulangi hingga jarak terpendek dari node awal ke seluruh node pada graf telah terhitung.

Contoh tersebut menunjukkan bagaimana algoritma Dijkstra dapat digunakan untuk menemukan jalur terpendek dalam sebuah graf. Pada contoh tersebut, pertama-tama dilakukan inisialisasi graf, node awal, dan nilai bobot pada setiap sisi. Kemudian, dilakukan pemrosesan graf dengan menggunakan algoritma Dijkstra. Hasil dari pemrosesan graf adalah daftar jarak terpendek dari node awal ke semua node yang ada pada graf.

B. Pseudocode Algoritma Dijkstra

- `Judul: Algoritma Dijkstra untuk Mencari Jalur Terpendek`

- `Deklarasi:`

1) `Variabel graph: representasi graf dengan daftar simpul dan jarak antar simpul`
2) `Variabel start_node: simpul awal dari pencarian jalur terpendek`

3) Variabel end_node: simpul tujuan dari pencarian jalur terpendek
4) Variabel unvisited: daftar simpul yang belum dikunjungi
5) Variabel distances: jarak terpendek yang diketahui dari simpul awal ke simpul yang dikunjungi
6) Variabel previous: simpul sebelumnya di jalur terpendek dari simpul awal ke simpul yang dikunjungi

❖ Implementasi:

1) Inisialisasi variabel distances dan previous dengan nilai awal. Setiap simpul memiliki jarak tak terbatas dari simpul awal dan tidak memiliki simpul sebelumnya di jalur terpendek.
2) Tambahkan simpul awal ke variabel unvisited.
3) Selama unvisited tidak kosong, lakukan langkah-langkah berikut:

   a. Pilih simpul dari unvisited dengan jarak terpendek dari simpul awal (ini adalah simpul pertama pada iterasi pertama). Letakkan simpul ini dalam variabel current_node.

   b. Untuk setiap simpul yang bertetangga dengan current_node, lakukan langkah-langkah berikut:

      - Hitung jarak baru dari simpul awal ke simpul ini melalui current_node.
      - Jika jarak baru lebih kecil dari distances yang diketahui sebelumnya, perbarui distances dan previous.
      - Jika simpul ini belum dikunjungi, tambahkan ke unvisited.

   c. Hapus current_node dari unvisited.

4.) Jalur terpendek dari simpul awal ke simpul tujuan dapat dikonstruksi dari previous.

C. Source Code Algoritma Dijkstra

```python
import heapq

def dijkstra(graf, start):
    jarak = {simpul: float('inf') for simpul in graf}
    jarak[start] = 0
    heap = [(0, start)]
    while heap:
        (jarak_saat_ini, simpul_saat_ini) = heapq.heappop(heap)
        if jarak_saat_ini > jarak[simpul_saat_ini]:
            continue
        for tetangga, bobot in graf[simpul_saat_ini].items():
            jarak_tetangga = jarak_saat_ini + bobot
            if jarak_tetangga < jarak[tetangga]:
                jarak[tetangga] = jarak_tetangga
                heapq.heappush(heap, (jarak_tetangga, tetangga))
    return jarak

graf = {'A': {'B': 3, 'D': 4},
        'B': {'A': 3, 'D': 2, 'E': 6},
        'C': {'E': 1},
        'D': {'A': 4, 'B': 2, 'E': 5},
        'E': {'B': 6, 'C': 1, 'D': 5}}

print(dijkstra(graf, 'A'))

```

Ouput

```
{'A': 0, 'B': 3, 'C': 10, 'D': 4, 'E': 9}
PS C:\Users\acer>
```

Penjelsan:

1. Impor library

```python
import heapq

```

2. Deklarasi fungsi dijkstra dengan parameter graf dan start. Variabel jarak diinisialisasi dengan nilai tak terhingga untuk setiap simpul dalam graf menggunakan dictionary comprehension.

```python
def dijkstra(graf, start):
    jarak = {simpul: float('inf') for simpul in graf}
    jarak[start] = 0
```

3. Variabel heap diinisialisasi sebagai list kosong. Setiap item dalam list heap akan berisi jarak dari simpul saat ini ke simpul lainnya dan simpul tujuan.

```
heap = [(0, start)]
```

4. Selama list heap tidak kosong, jalankan loop while. Ambil item pertama dari list heap dengan fungsi heappop dari heapq dan simpan dalam variabel jarak_saat_ini dan simpul_saat_ini.

```
while heap:
(jarak_saat_ini, simpul_saat_ini) = heapq.heappop(heap)
```

5. Jika jarak saat ini lebih besar dari jarak pada simpul saat ini, lanjutkan ke item selanjutnya dalam loop.

```
if jarak_saat_ini > jarak[simpul_saat_ini]:
continue
```

6. Loop for digunakan untuk mengeksplorasi semua tetangga dari simpul saat ini (simpul_saat_ini).

```
for tetangga, bobot in graf[simpul_saat_ini].items():
```

7. Hitung jarak baru dari simpul saat ini ke tetangga dengan menambahkan jarak saat ini ke bobot dari simpul saat ini ke tetangga.

```
jarak_tetangga = jarak_saat_ini + bobot
```

8. Jika jarak baru lebih kecil dari jarak sebelumnya pada simpul tetangga, perbarui jarak pada simpul tetangga dan tambahkan simpul tetangga dan jarak terbaru ke list heap dengan fungsi heappush dari heapq.

```
if jarak_tetangga < jarak[tetangga]:
    jarak[tetangga] = jarak_tetangga
    heapq.heappush(heap, (jarak_tetangga, tetangga))
```

9. Fungsi dijkstra mengembalikan dictionary jarak yang berisi jarak terpendek dari start ke setiap simpul dalam graf.

```
return jarak
```

10. Variabel graf didefinisikan sebagai dictionary yang berisi simpul-simpul dan bobot antar simpul dalam graf.

```
graf = {'A': {'B': 3, 'D': 4},
        'B': {'A': 3, 'D': 2, 'E': 6},
        'C': {'E': 1},
        'D': {'A': 4, 'B': 2, 'E': 5},
        'E': {'B': 6, 'C': 1, 'D': 5}}

print(dijkstra(graf, 'A'))
```

D. Analisis Kebutuhan Waktu algoritma Dijkstra

1. Analisis waktu algoritma Quicksort berdasarkan operasi/instruksi yang dieksekusi:

Berikut adalah analisis dari program tersebut berdasarkan operator penugasan atau assignment dan operator aritmatika yang digunakan:

1) Operator Penugasan (=): Digunakan untuk menginisialisasi variabel jarak dengan nilai infinity dan nilai jarak dari simpul start dengan nilai 0. Juga digunakan untuk memperbarui nilai jarak saat menemukan jalur yang lebih pendek dari simpul start ke simpul tetangga. Digunakan sebanyak 5 kali dalam program.
2) Operator Aritmatika (+): Digunakan untuk menghitung nilai jarak simpul tetangga saat mencari jalur terpendek dari simpul start ke simpul tetangga. Digunakan sebanyak 4 kali dalam program.

Total jumlah operasi abstrak atau operasi khas pada program di atas adalah 9 kali, yang terdiri dari 5 kali operator penugasan dan 4 kali operator aritmatika.

2.Analisis waktu algoritma Dijkstra berdasarkan operasi/instruksi yang dieksekusi

a. Inisialisasi jarak pada setiap simpul dalam graf: O(|V|)
b. Inisialisasi jarak start ke simpul lain dalam graf dengan nilai tak terbatas: O(|V|)
c. Inisialisasi heap dengan pasangan (0, start): O(log|V|)
d. Selama heap tidak kosong, eksekusi loop:
- Mengambil elemen teratas dari heap: O(log|V|)
- Jika nilai jarak saat ini lebih besar dari jarak di simpul saat ini, abaikan dan lanjutkan loop: O(1)
- Iterasi pada tetangga simpul saat ini, dan mengupdate jarak tetangga jika ditemukan jalur yang lebih pendek: O(|E|)
- Jika jarak tetangga lebih pendek, tambahkan pasangan jarak dan simpul tetangga ke heap: O(log|V|)

e. Kembalikan hasil jarak untuk setiap simpul: O(|V|)
f. Maka, total waktu eksekusi algoritma Dijkstra adalah O(|E|+|V|log|V|).

3. Analisis waktu algoritma Quicksort pada pendekatan best-case, worst-case, dan average-case:

1) Best-case scenario: kompleksitas waktu adalah O(1) karena hanya satu simpul yang diproses.
2) Worst-case scenario: kompleksitas waktu adalah O(E log V) karena semua simpul dan edge harus diproses.
3) Average-case scenario: kompleksitas waktu adalah O(E log V) karena diproses jumlah simpul dan edge yang sedang pada graf.

Dalam keseluruhan, kompleksitas waktu algoritma Dijkstra tergantung pada jumlah simpul dan edge dalam graf dan memiliki kompleksitas waktu O(E log V).

E. Referensi

1) Introduction to the Design & Analysis of Algorithms 3rd Edition karya Anany Levitin
2) https://pemburukode.com/mengenal-algoritma-aijkstra/

F. Lampiran Link Github

https://github.com/Maharanisafitri/TUGAS-ANALISIS-ALGORITMA