# *All-Pairs Shortest Path*

Kelompok B01 - sad B01

Fakultas Ilmu Komputer, Universitas Indonesia
{firman.hadi61,norman.bintang,reynaldo.wijaya,windi.chandra}@ui.ac.id

*All-pairs shortest path* (APSP) merupakan permasalahan yang mirip dengan *single-source shortest path* (SSSP) yang mana permasalahan tersebut adalah mencari jarak terpendek dari suatu *vertex* pada suatu graf ke setiap *vertex* lain pada graf tersebut. Permasalahan APSP sendiri adalah mencari jarak terpendek antara setiap pasang *vertex* pada graf. *Lecture notes* ini akan mengulas beberapa aspek dalam menyelesaikan permasalahan APSP, dengan asumsi bahwa pembaca sudah memahami tentang SSSP dan penyelesaiannya menggunakan algoritma BFS, Dijkstra dan Bellman-Ford.

Seluruh isi *lecture notes* ini merupakan inti dari bab *All-Pairs Shortest Path* pada buku Algorithms, karya Jeff Erickson[1].

## 1 Definisi Permasalahan

Diberikan suatu graf *weighted* $G = (V, E)$, untuk setiap pasang *vertex* $u$ dan $v$ pada $G$, kita ingin mencari informasi berikut:

- $dist(u, v)$, yaitu jarak (jumlah *weight* pada *path*) terpendek dari *vertex* $u$ ke $v$, dan
- $pred(u, v)$, yaitu *vertex* terakhir sebelum $v$ pada *path* dengan jarak terpendek dari *vertex* $u$ ke $v$.

Hasil yang diharapkan adalah *array* berukuran $|V| \times |V|$ untuk masing-masing nilai $dist(u, v)$ dan $pred(u, v)$. Lebih rincinya, nilai *dist* dan *pred* didefinisikan sebagai berikut untuk beberapa kasus:

- Apabila tidak ada *path* dari *vertex* $u$ ke $v$, maka $dist(u, v) = \infty$ dan $pred(u, v) = \text{NULL}$.
- Apabila terdapat *negative cycle* (*cycle* yang menghasilkan jumlah *weight* negatif) pada suatu *path* dari *vertex* $u$ ke *vertex* $v$, maka $dist(u, v) = -\infty$ dan $pred(u, v) = \text{NULL}$.
- Apabila tidak terdapat *negative cycle* pada suatu *path* dari *vertex* $u$ ke dirinya sendiri, maka $dist(u, u) = 0$ dan $pred(u, u) = \text{NULL}$.

Setelah mendefinisikan permasalahan APSP, maka kita bisa melanjutkan ke cara menyelesaikan permasalahan tersebut. Beberapa cara penyelesaian yang dijelaskan di bawah akan fokus ke nilai *dist* terlebih dahulu. Nilai *pred* sebenarnya bisa didapat dengan sedikit modifikasi pada algoritma-algoritma di bawah. Dapat ditunjukkan juga bahwa nilai *pred* dapat dicari apabila telah diketahui nilai *dist* untuk setiap pasang *vertex*. Cara tersebut akan ditunjukkan pada bagian akhir.

## 2 SSSP untuk Menyelesaikan APSP

Untuk beberapa graf dengan sifat-sifat tertentu, kita dapat memanfaatkan algoritma SSSP untuk menyelesaikan permasalahan ini. Penjelasannya cukup mudah, karena SSSP mencari *shortest path* dari satu *vertex* ke semua *vertex* lainnya, maka kita dapat gunakan algoritma SSSP dari setiap *vertex* yang ada. Berikut adalah algoritma yang dapat digunakan berdasarkan sifat graf yang dimiliki:

**Table 1.** Pendekatan SSSP untuk setiap graf [1]

| Sifat graf | Algoritma | Kompleksitas Waktu |
|---|---|---|
| *unweighted* | BFS | $O(\lvert V\rvert \times (\lvert V\rvert + \lvert E\rvert)) = O(\lvert V\rvert^3)$ |
| *directed, acyclic* | DFS | $O(\lvert V\rvert \times (\lvert V\rvert + \lvert E\rvert)) = O(\lvert V\rvert^3)$ |
| *edge* nonnegatif | Dijkstra [2] | $O(\lvert V\rvert \times \lvert E\rvert \log \lvert V\rvert) = O(\lvert V\rvert^3 \log \lvert V\rvert)$ |
| *cycle* nonnegatif | Bellman-Ford | $O(\lvert V\rvert \times \lvert V\rvert\lvert E\rvert) = O(\lvert V\rvert^4)$ |

Terlihat bahwa menyelesaikan APSP menggunakan SSSP memberikan setidaknya $O(|V|^3)$ untuk graf-graf dengan sifat tertentu. Berdasarkan tabel di atas, terlihat bahwa APSP semakin 'sulit' dikarenakan adanya *edge* dengan *weight* negatif (untuk selanjutnya akan disebut dengan *edge* negatif).

## 3 Algoritma Johnson

### 3.1 *Reweighting* untuk menangani *edge* negatif

*Reweighting* merupakan teknik untuk mendefinisikan ulang suatu *weight* pada *edge* $u \to v$. Namun, metode *reweighting* yang tidak tepat bisa mengubah *shortest path* asli dari suatu graf. Berikut adalah metode *reweighting* yang digunakan oleh Donald Johnson pada algoritmanya (yang akan dijelaskan selanjutnya) pada tahun 1973. Misalkan untuk setiap *node* $v$ diberikan suatu nilai $\pi(v)$ (bisa positif, negatif, atau nol), fungsi *weight* baru pada *edge* $w'(u \to v)$ didefinisikan sebagai berikut:

$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$

Maka, untuk setiap *path* $u \Rightarrow v$ (dibaca dari $u$ ke $v$), maka total *weight*-nya adalah:

$$w'(u \Rightarrow v) = \pi(u) + w(u \Rightarrow v) - \pi(v)$$

Definisi ulang *weight* tersebut tidak akan mengubah *shortest path* karena untuk setiap kemungkinan *path* $u \Rightarrow v$, total dari *weight path* tersebut akan

[1] asumsi $|E| = O(|V|^2)$
[2] dengan implementasi *binary heap*

ditambah dengan nilai yang sama (yaitu $\pi(u) - \pi(v)$) sehingga tidak akan mengubah *path* dengan jarak terpendek. Namun, perlu diperhatikan bahwa apabila kita mengurutkan *shortest path* dari setiap kemungkinan *node* $u$ dan $v$, urutan tersebut bisa jadi berubah.

Dengan definisi tersebut, kita tinggal menentukan nilai $\pi$ untuk setiap *node* agar *weight edge* yang baru menjadi nonnegatif. Salah satu caranya adalah menggunakan *shortest path* dari suatu *node* yang bisa mencapai ke seluruh *node* yang lain. Misalkan pada graf terdapat suatu *node* $s$ yang bisa mencapai semua *node* lain. Definisikan $\pi(u) = dist(s, u)$ yaitu *weight* pada *shortest path* dari $s \Rightarrow u$ sehingga *weight* yang baru adalah:

$$w'(u \to v) = dist(s, u) + w(u \to v) - dist(s, v)$$

**Theorem 1.** *Jika tidak ada negative cycle pada graf G, maka untuk setiap edge* $u \to v$, $w'(u \to v) \geq 0$. *(weight edge yang baru pasti nonnegatif)*

*Proof.* Misalkan bahwa $w'(u \to v) < 0$. Akibatnya, $dist(s, u) + w(u \to v) < dist(s, v)$. Hal ini menandakan bahwa terdapat *path* $s \Rightarrow v$ yang mempunyai *weight* lebih pendek daripada $dist(u, v)$ yaitu dengan melalui *node* $u$. Hal ini berkontradiksi dengan definisi $dist(s, v)$ sendiri sebagai jarak terpendek dari $s \Rightarrow v$ sehingga $w'(u \to v) \geq 0$ haruslah benar.

Berdasarkan teorema tersebut, terlihat bahwa dengan metode *reweighting* seperti di atas akan mengeliminasi *edge* negatif untuk graf tanpa *negative cycle*.

### 3.2 Algoritma

Algoritma Johnson memanfaatkan metode *reweighting* yang telah dijelaskan sebelumnya. Pertama, akan dihitung $dist(s, v)$ untuk setiap *node* $v$ dan suatu *node* $s$ yang dapat mencapai setiap *node* lainnya. Apabila tidak ada *node* $s$ yang memenuhi syarat, maka akan dibuat *dummy node* yang mempunyai *edge* secara langsung ke setiap *node* lain, dengan *weight* nol (node lain tidak boleh bisa mencapai *dummy node* agar *shortest path* tidak berubah). Perhitungan $dist(s, v)$ dapat dilakukan dengan algoritma Bellman-Ford. Sebagai tambahan, algoritma Bellman-Ford dapat mendeteksi adanya *negative cycle*, dan algoritma Johnson akan berhenti (gagal) apabila terdapat *negative cycle*.

Kedua, dengan memanfaatkan $dist(s, v)$, maka kita bisa mendapatkan *weight edge* yang baru. Untuk setiap *node* $u$, gunakan *weight* yang baru untuk menghitung SSSP dari $u$. Karena sekarang seluruh *weight edge* nonnegatif, kita bisa gunakan algoritma Dijkstra untuk mendapatkan *shortest path* dari setiap pasang *node* (dengan *weight edge* yang baru). Untuk mendapatkan *shortest path* dengan *weight* yang asli, manfaatkan definisi *weight* yang baru dan gunakan manipulasi aljabar:

$$w'(u \Rightarrow v) = dist(s,u) + w(u \Rightarrow v) - dist(s,v)$$
$$w(u \Rightarrow v) = w'(u \Rightarrow v) - dist(s,u) + dist(s,v)$$

Maka, algoritma Johnson dapat ditulis seperti berikut:

---

**Algoritma 1:** Algoritma Johnson untuk APSP

---

**masukan:** graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
**keluaran:** $dist[u][v]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$

```
add a new node s
foreach v ∈ V do
|  add a new edge s → v with w(s → v) = 0
end

dist[s][·] ← Bellman-Ford(G, s, w)
if Bellman-Ford finds a negative cycle then
|  stop (fail)
end

foreach u → v ∈ E do
|  w'(u → v) ← dist[s][u] + w(u → v) − dist[s][v]
end

foreach u ∈ v do
|  dist'[u][·] ← Dijkstra(G, u, w')
end

foreach u ∈ V do
|  foreach v ∈ V do
|  |  dist[u][v] = dist'[u][v] − dist[s][u] + dist[s][v]
|  end
end

return dist without s
```

---

Secara keseluruhan, kompleksitas waktu algoritma Johnson didominasi oleh pemanggilan algoritma Dijkstra sebanyak $|V|$ kali sehingga kompleksitas waktunya adalah $O(|V| \times |E| \log |V|) = O(|V|^3 \log |V|)$. Terlihat bahwa algoritma ini lebih efisien dibandingkan menggunakan algoritma Bellman-Ford sebanyak $|V|$ kali.

## 4 Pendekatan Rekursif

Pada bagian ini, akan selalu diasumsikan graf tidak mempunyai *negative cycle*.

### 4.1 *Dynamic Programming*

Secara umum, APSP juga bisa diselesaikan dengan pendekatan *dynamic programming*. Untuk graf yang sifatnya DAG (*Directed Acyclic Graph*), $dist(u, v)$ dapat didefinisikan secara rekursif sebagai berikut:

$$dist(u,v) = \begin{cases} 0 & \text{jika } u = v \\ \min\limits_{x \to v} \left(dist(u,x) + w(x \to v)\right) & \text{jika tidak} \end{cases}$$

Apabila graf tidak bersifat DAG, kita bisa memanfaatkan observasi yang digunakan pada algoritma Bellman-Ford, yaitu *shortest path* $u \Rightarrow v$ pasti melewati paling banyak $(|V|-1)$ *edge*. Dengan demikian, misalkan $dist(u, v, l)$ adalah panjang *shortest path* $u \Rightarrow v$ dengan menggunakan paling banyak $l$ *edge*, maka $dist(u, v, l)$ dapat didefinisikan secara rekursif sebagai berikut:

$$dist(u,v,l) = \begin{cases} 0 & \text{jika } l = 0 \text{ dan } u = v \\ \infty & \text{jika } l = 0 \text{ dan } u \neq v \\ \min \begin{Bmatrix} dist(u,v,l-1) \\ \min\limits_{x \to v} \left(dist(u,x,l-1) + w(x \to v)\right) \end{Bmatrix} & \text{jika tidak} \end{cases}$$

sehingga $dist(u, v) = dist(u, v, |V| - 1)$.

Rekurensi diatas dapat diubah menjadi algoritma *Dynamic Programming* (DP) dengan mudah. Algoritma tersebut dirancang pertama kali oleh Alfonso Shimbel pada tahun 1954. Berikut adalah *pseudocode* algoritma Shimbel:

**Algoritma 2:** Algoritma Shimbel

**masukan:** graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
**keluaran:** $dist[u][v][l]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak $l$ *edge*

```
foreach u ∈ V do
    foreach v ∈ V do
        if u = v then dist[u][v][0] ← 0
        else dist[u][v][0] ← ∞
    end
end

foreach l ← 1 to |V| − 1 do
    foreach u ∈ V do
        foreach v ∈ V, v ≠ u do
            dist[u][v][l] ← dist[u][v][l − 1]
            foreach (x → v) ∈ E do
                dist[u][v][l] ← min(dist[u][v][l], dist[u][x][l−1] + w(x → v))
            end
        end
    end
end

return dist
```

Perhatikan bahwa pada saat kita melakukan *for loop* untuk setiap *edge* $(x \to v)$, terdapat paling banyak $|V|$ *edge* dari setiap *node* $x$ ke *node* $v$ (dengan asumsi pada graf tidak ada lebih dari satu *edge* yang menghubungkan dua *node* yang sama) sehingga kompleksitas waktu yang didapat adalah $O(|V| \times |V| \times |V| \times |V|) = O(|V|^4)$.

Observasi lebih lanjut menunjukkan bahwa kita dapat melakukan efisiensi penggunaan memori dengan mengeliminasi indeks $l$ seperti algoritma Bellman-Ford, yang mana pada akhirnya improvisasi algoritma ini menjadi mirip dengan menjalankan algoritma Bellman-Ford untuk setiap *node* pada graf:

**Algoritma 3:** Algoritma Shimbel, ditambah dengan efisiensi penggunaan memori

**masukan:** graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
**keluaran:** $dist[u][v]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$

```
foreach u ∈ V do
    foreach v ∈ V do
        if u = v then dist[u][v] ← 0
        else dist[u][v] ← ∞
    end
end

foreach l ← 1 to |V| − 1 do
    foreach u ∈ V do
        foreach (x → v) ∈ E do
            dist[u][v] ← min(dist[u][v], dist[u][x] + w(x → v))
        end
    end
end

return dist
```

### 4.2 Mempercepat dengan *Divide and Conquer*

Michael Fischer dan Albert Meyer pada tahun 1971 mem-*propose* efisiensi dari algoritma DP di atas dengan pendekatan *divide and conquer*. Sebelumnya, algoritma rekursif memecah pencarian *shortest path* menjadi *shortest path* yang lebih pendek, ditambah satu *edge*. Kita juga bisa memecah *shortest path* menjadi dua pencarian *shortest path* dengan mencari *node* tengah diantara *path* tersebut sehingga kedua *shortest path* menggunakan banyak *edge* yang sama: (untuk mempermudah, asumsi $w(v \to v) = 0$ untuk setiap *node* $v$)

$$dist(u, v, l) = \begin{cases} w(u \to v) & \text{jika } l = 1 \\ \min\limits_{x}(dist(u, x, \frac{l}{2}) + dist(x, v, \frac{l}{2})) & \text{jika tidak} \end{cases}$$

Sekilas kita dapat melihat bahwa rumus rekursif hanya terdefinisi untuk $l$ yang merupakan bilangan dua pangkat. Untungnya, hal ini tidak menjadi masalah karena $dist(u, v, l) = dist(u, v, l + 1)$ untuk $l \geq |V| - 1$ (ingat kembali bahwa *shortest path* paling banyak menggunakan $|V| - 1$ *edge*). Maka dari itu, kita dapat mendefinisikan $dist(u, v) = dist(u, v, l)$ dengan $l = 2^{\lceil \lg |V| \rceil}$. Terlebih lagi, kita cukup memperhatikan $\lg |V|$ kemungkinan nilai $l$. Berikut adalah *pseudocode* algoritma Fischer-Meyer, $dist[u][v][i]$ adalah nilai dari $dist(u, v, 2^i)$.

**Algoritma 4:** Algoritma Fischer-Meyer untuk APSP

```
masukan: graf weighted G = (V, E) dan fungsi weight w(u → v)
keluaran: dist[u][v][i] = panjang shortest path u ⇒ v untuk setiap
          pasang node u, v ∈ V dengan menggunakan paling banyak 2^i
          edge (0 ≤ i ≤ ⌈lg |V|⌉)

foreach u ∈ V do
    foreach v ∈ V do
        dist[u][v][0] ← w(u → v)
    end
end

foreach i ← 1 to ⌈lg |V|⌉ do
    foreach u ∈ V do
        foreach v ∈ V do
            dist[u][v][i] ← ∞
            foreach x ∈ V do
                dist[u][v][i] ←
                  min(dist[u][v][i], dist[u][x][i − 1] + dist[x][v][i − 1])
            end
        end
    end
end

return dist
```

Terlihat bahwa algoritma di atas memiliki kompleksitas waktu $O(|V|^3 \log |V|)$ dan kompleksitas memori $O(|V|^2 \log |V|)$.

Algoritma tersebut masih bisa diperbaiki kompleksitas memorinya dengan menghapus dimensi ketiga dari $dist[u][v][i]$ sehingga tersisa $dist[u][v]$ saja, dan kompleksitas memorinya menjadi $O(|V|^2)$. Hal tersebut ditunjukkan oleh Leyzorek et al. pada tahun 1957:

**Algoritma 5:** Algoritma Leyzorek untuk APSP

**masukan:** graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
**keluaran:** $dist[u][v]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$

```
foreach u ∈ V do
    foreach v ∈ V do
        dist[u][v] ← w(u → v)
    end
end

foreach i ← 1 to ⌈lg |V|⌉ do
    foreach u ∈ V do
        foreach v ∈ V do
            foreach x ∈ V do
                dist[u][v] ← min(dist[u][v], dist[u][x] + dist[x][v])
            end
        end
    end
end

return dist
```

### 4.3 Menganalogikan dengan Perkalian Matriks (*Funny Matrix Multiplication*)

Ingat kembali algoritma perkalian matriks, lebih spesifiknya pada matriks persegi:

**Algoritma 6:** Perkalian matriks persegi

**masukan:** Matriks persegi $A$ dan $B$, masing-masing berukuran $n \times n$
**keluaran:** Matriks persegi $AB$ yang merupakan hasil $A \times B$

```
foreach i ← 1 to n do
    foreach j ← 1 to n do
        AB[i][j] ← 0
        foreach k ← 1 to n do
            AB[i][j] ← AB[i][j] + A[i][k] × B[k][j]
        end
    end
end

return AB
```

Sekarang, tinjau *inner loop* pada algoritma Fischer-Meyer berikut untuk setiap nilai $0 \leq i \leq \lceil \lg |V| \rceil$:

**Algoritma 7:** *Inner loop* pada algoritma Fischer-Meyer

```
foreach u ∈ V do
    foreach v ∈ V do
        dist[u][v][i] ← ∞
        foreach x ∈ V do
            dist[u][v][i] ←
              min(dist[u][v][i], dist[u][x][i − 1] + dist[x][v][i − 1])
        end
    end
end
```

Kedua *pseudocode* di atas mempunyai pola yang mirip. Konstruksi nilai $dist[\cdot][\cdot][i]$ dapat dianalogikan sebagai 'mengkuadratkan' nilai $dist[\cdot][\cdot][i-1]$, dengan analogi-analogi sebagai berikut:

- Inisialisasi nilai $dist[\cdot][\cdot][i]$ dengan $\infty$ dianalogikan sebagai inisialisasi nilai elemen matriks dengan nol.
- *Update* nilai $dist[\cdot][\cdot][i]$ dianalogikan sebagai *update* nilai elemen matriks pada algoritma pengkuadratan matriks.

Operasi 'perkalian' ini seringkali disebut perkalian matriks *min-plus*, atau *funny/distance matrix multiplication*. Perkalian matriks *min-plus* bisa dikatakan sebagai perkalian matriks dengan operasi penjumlahan skalar diganti dengan operasi min, dan operasi perkalian skalar diganti dengan operasi penjumlahan.

Definisikan $A \bigotimes B$ sebagai matriks $A$ dikali dengan $B$ dengan perkalian matriks *min-plus*. Berikut adalah *pseudocode* dari algoritma perkalian matriks *min-plus* untuk matriks persegi, perhatikan kesamaannya dengan *inner loop* dari algoritma Fischer-Meyer di atas.

**Algoritma 8:** Algoritma perkalian *min-plus* untuk matriks persegi

```
masukan: Matriks persegi A dan B, masing-masing berukuran n × n
keluaran: Matriks persegi AB yang merupakan hasil A ⊗ B
foreach i ← 1 to n do
    foreach j ← 1 to n do
        AB[i][j] ← ∞
        foreach k ← 1 to n do
            AB[i][j] ← min(AB[i][j], A[i][k] + B[k][j])
        end
    end
end

return AB
```

Lebih lanjut lagi, sebenarnya langkah-langkah algoritma yang memanfaatkan *dynamic programming* sebelumnya dapat kita tulis ulang menggunakan operasi $\bigotimes$.

**Algoritma Shimbel** : Perhatikan bahwa *loop for* pertama berguna untuk inisialisasi matriks $dist[\cdot][\cdot][0]$ (yang dikatakan sebagai matriks identitas untuk operator perkalian $\bigotimes$), kemudian untuk *loop for* selanjutnya, mendapatkan $dist[\cdot][\cdot][l]$ sebenarnya sama saja dengan menghitung $dist[\cdot][\cdot][l-1] \bigotimes dist[\cdot][\cdot][1]$ yang mana $dist[\cdot][\cdot][1]$ merupakan *adjacency matrix*. Pada akhirnya, jawaban APSP yang ingin dicari berada pada $dist[\cdot][\cdot][|V|-1]$ yang merupakan hasil perkalian $\bigotimes$ sebanyak $|V|-1$ kali.

**Algoritma Fischer-Meyer** : Algoritma Fischer-Meyer memanfaatkan sifat asosiatif dari perkalian matriks[3]. Perhitungan $dist[\cdot][\cdot][2^i]$ sebenarnya

[3] Perkalian matriks *min-plus* bersifat asosiatif karena operasi min sebagai penjumlahan dan operasi + sebagai perkalian pada bilangan riil membentuk sebuah *semiring*, perkalian matriks dalam *ring*/*semiring* bisa dipastikan mempunyai sifat asosiatif (bisa dibuktikan[2]), hal ini dibahas dalam aljabar abstrak (*tropical mathematics*[3]) dan tidak akan dibahas lebih lanjut pada *lecture notes* ini.

dilakukan dengan meng'kuadrat'kan $dist[\cdot][\cdot][2^{i-1}]$ (dengan kata lain, $dist[\cdot][\cdot][2^{i}] = dist[\cdot][\cdot][2^{i-1}] \bigotimes dist[\cdot][\cdot][2^{i-1}]$). Teknik ini sama seperti teknik *exponentiation by squaring*.

Dengan demikian, mungkin kita bisa mencoba untuk mempercepat perkalian matriks *min-plus*. Sebelumnya, telah dibahas bahwa perkalian matriks standar pada matriks berukuran $n \times n$ dapat dipercepat menggunakan teknik *divide and conquer* seperti algoritma Strassen yang mempunyai kompleksitas waktu $O(n^{\lg 7})$. Sayangnya, algoritma Strassen tidak bisa diaplikasikan ke perkalian matriks *min-plus* karena algoritma Strassen menggunakan pengurangan (invers dari penjumlahan) dalam mengalikan matriks, dan operasi min tidak mempunyai invers.

Selanjutnya, sebenarnya terdapat sebuah algoritma untuk mencari APSP dari graf yang bersifat *unweighted* dan *undirected* yang memanfaatkan perkalian matriks standar. Algoritma tersebut di-*propose* oleh Raimund Seidel yang mempunyai kompleksitas waktu *expected* yaitu $O(M(|V|) \log |V|)$, dengan $M(n) = O(n^{2.37})$ yang merupakan waktu yang dibutuhkan untuk mengalikan matriks $n \times n$ secara standar.

### 4.4 Formulasi DP yang Lain (Algoritma Floyd-Warshall)

Warshall (Roy dan Kleene juga) mem-*propose* cara rekursif yang mirip, namun berbeda pada parameter ketiganya. Alih-alih membatasi banyaknya *edge* paling banyak yang bisa digunakan, mereka membatasi *node* yang bisa dilewati. Dilewati disini berarti "masuk dan keluar". Sebagai contoh, *path* $w \to x \to y \to z$ berarti dimulai dari $w$, melewati $x$ dan $y$, dan berakhir pada $z$ (*node* awal dan akhir tidak dihitung sebagai melewati). Misalkan setiap *node* dinomori 1 sampai $|V|$ (bebas), definisikan $\pi(u, v, r)$ sebagai *shortest path* dari $u$ ke $v$ yang hanya melewati *node* yang bernomor kurang dari atau sama dengan $r$.

Dengan definisi tersebut, struktur rekursif yang didapat adalah:

– *Path* $\pi(u, v, 0)$ tidak bisa melewati *node* apapun, sehingga pasti menggunakan *edge* $u \to v$ (bila ada).

– Untuk $r > 0$, $\pi(u, v, r)$ mempunyai dua pilihan, yaitu melewati *node* bernomor $r$, atau tidak sama sekali.
  - Apabila melewati *node* bernomor $r$, maka *path* tersebut bisa dipecah menjadi $u \Rightarrow r$ dan $r \Rightarrow v$. Kedua *path* pasti melewati node yang mempunyai nomor kurang dari sama dengan $r - 1$. Maka, *path* tersebut bisa dipecah menjadi $\pi(u, r, r - 1)$ dan $\pi(r, v, r - 1)$.
  - Apabila tidak, maka *path*-nya sama dengan $\pi(u, v, r - 1)$.

Berdasarkan kedua pilihan di atas, $\pi(u, v, r)$ pasti merupakan *path* terpendek dari dua pilihan di atas.

Misalkan $dist(u, v, r)$ adalah panjang dari *path* $\pi(u, v, r)$. Berdasarkan struktur rekursif di atas, maka $dist(u, v, r)$ mempunyai rekurens berikut:

$$dist(u, v, r) = \begin{cases} w(u \to v) & \text{jika } r = 0 \\ \min \begin{Bmatrix} dist(u, v, r - 1) \\ dist(u, r, r - 1) + dist(r, v, r - 1) \end{Bmatrix} & \text{jika tidak} \end{cases}$$

Dengan demikian, dapat dilihat bahwa $dist(u, v)$ yang ingin kita cari adalah $dist(u, v, |V|)$. Rekurensi di atas dapat dihitung dengan *dynamic programming* yang sederhana dengan kompleksitas waktu $O(|V|^3)$, seperti *pseudocode* berikut:

**Algoritma 9:** Algoritma Kleene untuk APSP

**masukan:** graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
**keluaran:** $dist[u][v][r]$ = panjang *shortest path* $u \Rightarrow v$ yang harus melewati *node* bernomor kurang dari atau sama dengan $r$, untuk setiap pasang *node* $u, v \in V$

```
foreach u ∈ V do
    foreach v ∈ V do
        dist[u][v][0] ← w(u → v)
    end
end

foreach r ← 1 to |V| do
    foreach u ∈ V do
        foreach v ∈ V do
            dist[u][v][r] ←
              min(dist[u][v][r − 1], dist[u][r][r − 1] + dist[r][v][r − 1])
        end
    end
end

return dist
```

Kemudian, seperti algoritma-algoritma sebelumnya, kita dapat membuang indeks ketiga dari *array* *dist* menjadi seperti berikut:

---

**Algoritma 10:** Algoritma Floyd-Warshall untuk APSP

---

**masukan:** graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
**keluaran:** $dist[u][v]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$

**foreach** $u \in V$ **do**
  **foreach** $v \in V$ **do**
    $dist[u][v] \leftarrow w(u \to v)$
  **end**
**end**

**foreach** $r \leftarrow 1$ to $|V|$ **do**
  **foreach** $u \in V$ **do**
    **foreach** $v \in V$ **do**
      $dist[u][v] \leftarrow \min(dist[u][v], dist[u][r] + dist[r][v])$
    **end**
  **end**
**end**

**return** *dist*

---

Algoritma di atas mempunyai kompleksitas waktu $O(|V|^3)$ dan kompleksitas memori $O(|V|^2)$.

# 5 Menangani *Negative Cycle*

Algoritma-algoritma sebelumnya tidak selalu menghasilkan nilai *dist* yang benar apabila terdapat *negative cycle* pada graf. Bagaimana apabila kita membutuhkan APSP pada graf yang memiliki *negative cycle*?

Untungnya, kita bisa melakukan modifikasi pada algoritma Floyd-Warshall untuk menangani *negative cycle*. Hal menarik yang terlihat setelah menjalankan algoritma Floyd-Warshall adalah apabila graf mempunyai *negative cycle*, pasti terdapat *node* $u$ dengan $dist[u][u] < 0$ yang artinya terdapat *negative cycle* yang melalui *node* $u$. Kita bisa memanfaatkan fakta ini untuk mencari *path* $u \Rightarrow v$ mana saja yang bisa mengandung *negative cycle*.

Idenya adalah untuk setiap pasang *node* $(u, v)$, kita mencoba setiap *node* $x$ dengan $dist[x][x] < 0$. Apabila $dist[u][x] < \infty$ dan $dist[x][v] < \infty$, maka kita bisa membuat *path* $u \Rightarrow v$ yang melewati $x$. Karena kita tahu *node* $x$ merupakan bagian dari suatu *negative cycle*, maka kita dapat pastikan $dist(u, v) = -\infty$ (karena bisa melewati *negative cycle* dari $x$). Berikut adalah *pseudocode* untuk algoritma di atas.

**Algoritma 11:** Prosedur tambahan untuk menangani *negative cycle* pada algoritma Floyd-Warshall[4]

```
masukan: graf weighted G = (V, E) dan fungsi weight w(u → v)
keluaran: dist[u][v] = panjang shortest path u ⇒ v untuk setiap pasang
          node u, v ∈ V

dist[·][·] ← Floyd-Warshall(G, w)
foreach x ∈ V do
    if dist[x][x] ≥ 0 then
        continue
    end

    foreach u ∈ V do
        foreach v ∈ V do
            if dist[u][x] < ∞ & dist[x][v] < ∞ then
                dist[u][v] ← −∞
            end
        end
    end
end

return dist
```

Dapat terlihat bahwa kompleksitas waktu algoritma tersebut adalah $O(|V|^3)$ sehingga tidak akan mengubah kompleksitas waktu algoritma keseluruhan apabila kita menggunakan algoritma Floyd-Warshall.

## 6 Menghitung *pred* dengan Hasil *dist*

Algoritma-algoritma yang telah dijelaskan sebelumnya hanya menghasilkan nilai *dist*, sedangkan terkadang kita membutuhkan nilai *pred* untuk mendapatkan *shortest path*-nya, tidak hanya panjangnya saja. Untungnya, apabila kita telah mengetahui $dist[\cdot][\cdot]$, kita bisa mengetahui nilai $pred[\cdot][\cdot]$ dengan mudah. *Pseudocode* algoritma berikut menghasilkan nilai $pred[\cdot][\cdot]$, apabila diketahui $dist[\cdot][\cdot]$:

**Algoritma 12:** Algoritma untuk mencari nilai *pred*

```
masukan: graf weighted G = (V, E), fungsi weight w(u → v), dan
         dist[·][·] untuk graf G
keluaran: pred[u][v] = node yang dilewati tepat sebelum node v pada
          shortest path u ⇒ v, untuk setiap pasang node u, v ∈ V

foreach u ∈ V do
    foreach v ∈ V do
        pred[u][v] ← NULL
    end
end

foreach (x → v) ∈ E do
    foreach u ∈ V do
        if dist[u][v] ≠ ∞ and dist[u][v] ≠ −∞ then
            if dist[u][x] + w(x → v) = dist[u][v] then
                pred[u][v] ← x
            end
        end
    end
end

return pred
```

Ide dari algoritma di atas cukup sederhana, kita ingin mencari untuk setiap *edge* $x \to v$, *shortest path* $u \Rightarrow v$ mana saja yang menggunakan *edge* $x \to v$ untuk mencapai $v$. Jika $dist[u][v] = \infty$, maka bisa dikatakan tidak ada path $u \Rightarrow v$ sehingga $pred[u][v] = \text{NULL}$. Demikian juga pada kasus $dist[u][v] = -\infty$ (terdapat *negative cycle*). Apabila tidak, jika $dist[u][x] + w(x \to v) = dist[u][v]$, maka dapat dikatakan *shortest path* $u \Rightarrow x$ kemudian ditambah *edge* $x \to v$ akan memberikan *shortest path* $u \Rightarrow v$.

Kompleksitas waktu algoritma tersebut adalah $O(|E||V|) = O(|V|^3)$ sehingga tidak akan mempengaruhi kompleksitas waktu algoritma keseluruhan dalam menyelesaikan APSP apabila kita menggunakan Floyd-Warshall, Johnson, maupun algoritma apapun yang dijelaskan di atas. Dapat dilihat dengan mudah bahwa kompleksitas memori yang digunakan juga efisien yaitu $O(|V|^2)$ yang hanya untuk menyimpan jawaban.

## 7 Kesimpulan

Tabel berikut menunjukkan kompleksitas waktu algoritma yang telah ditunjukkan di atas:

**Table 2.** Algoritma APSP (yang telah dijelaskan) dan kompleksitas waktunya

| Algoritma | Kompleksitas Waktu |
|---|---|
| Johnson | $O(\|V\|\|E\| \log \|V\|) = O(\|V\|^3 \log \|V\|)$ |
| Shimbel | $O(\|V\|^4)$ |
| Fischer-Meyer | $O(\|V\|^3 \log \|V\|)$ |
| Floyd-Warshall | $O(\|V\|^3)$ |

Berdasarkan tabel di atas, terlihat bahwa satu-satunya algoritma di sini yang bergantung pada banyaknya *edge* adalah algoritma Johnson. Algoritma Johnson cocok diterapkan pada graf yang bersifat *sparse* (*edge*-nya sedikit, $|E| \ll |V|^2$) dan algoritma Floyd-Warshall tidak bergantung pada banyak *edge* sehingga mempunyai performa yang sama pada tipe graf apapun. Kita bisa memilih algoritma yang coock (diantara dua pilihan tersebut) apabila kita tahu sifat graf yang akan kita cari APSP-nya. Untuk algoritma Shimbel maupun Fischer-Meyer, mungkin bisa diaplikasikan ke permasalahan yang lebih spesifik seperti mencari APSP namun dengan menggunakan paling banyak $k$ *edge*, dan lain-lain.

Setelah mendapatkan nilai *dist* untuk setiap pasang *node*, maka kita bisa menggunakan algoritma di bagian sebelumnya untuk mencari nilai *pred*. Algoritma tersebut juga akan berjalan lebih cepat pada graf yang banyak *edge*-nya lebih sedikit apabila dibandingkan dengan graf dengan banyak *node* yang sama, namun *edge*-nya lebih banyak (lebih *dense*).

Menyelesaikan APSP melalui perspektif perkalian matriks menunjukkan bahwa beberapa permasalahan bisa diselesaikan melalui banyak cara. Bisa saja dengan memandang permasalahan secara berbeda, kita bisa melakukan optimisasi yang berbeda sehingga mencapai solusi yang lebih cepat. Contohnya pada algoritma Seidel untuk graf *unweighted* dan *undirected* yang sebenarnya memanfaatkan optimisasi perkalian matriks sehingga mencapai kompleksitas waktu yang lebih efisien dibandingkan melakukan BFS dari setiap *node*.

Algoritma-algoritma di atas juga belum tentu menghasilkan jawaban yang benar apabila terdapat *negative cycle* pada graf. Apabila graf memiliki *negative cycle*, perlu dilakukan modifikasi khusus untuk algoritmanya. Perlu diketahui juga, belum ditemukan algoritma dengan kompleksitas waktu $O(|V|^{3-\epsilon})$ dengan $\epsilon > 0$ untuk graf yang mempunyai sifat umum, jadi algoritma Floyd-Warshall adalah solusi terbaik (sampai saat ini) yang bisa dilakukan apabila kita tidak mengetahui sifat khusus dari graf yang ingin kita cari APSP-nya.

## Referensi


1. Erickson, J. (2019). Algorithms.
2. Matrix Multiplication is Associative. Diakses 22 November 2019, https://proofwiki.org/wiki/Matrix_Multiplication_is_Associative
3. Speyer, D., & Sturmfels, B. (2009). Tropical mathematics. Mathematics Magazine, 82(3), 163-173.
4. CP-Algorithms. Finding a negative cycle in a graph. Diakses 27 November 2019. https://cp-algorithms.com/graph/finding-negative-cycle-in-graph.html