All-Pairs Shortest Path

sad B01

sad B01

Fakultas Ilmu Komputer
Universitas Indonesia

Firman Hadi P. (1606862721)

Norman Bintang (1606862772)

Reynaldo Wijaya H. (1706028625)

Windi Chandra (1606862785)

All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut.

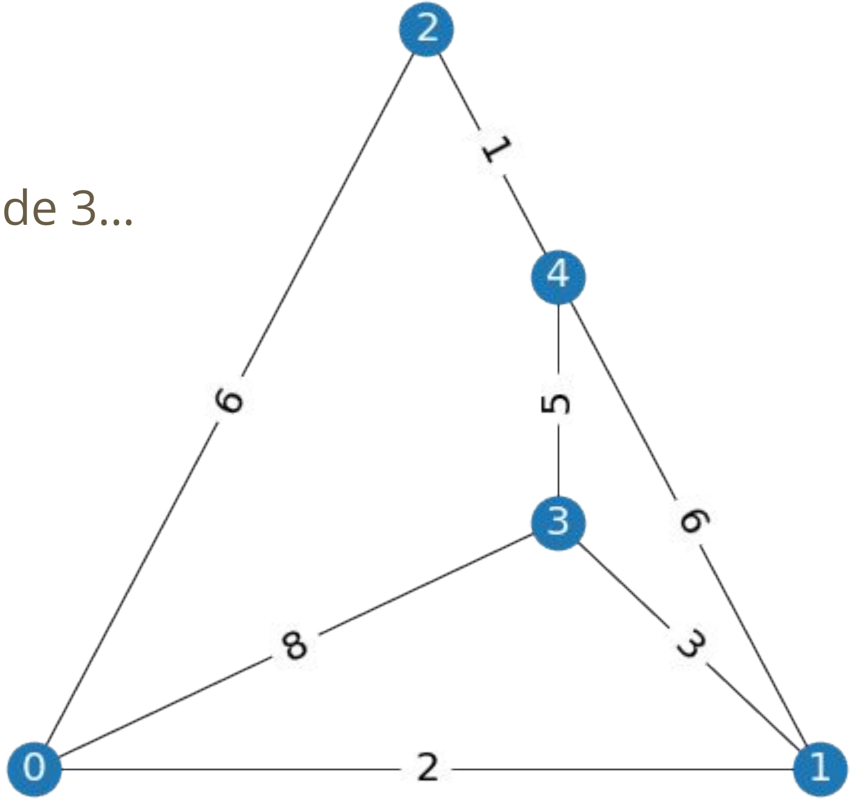
Kita ingin mengetahui *shortest path* dari node 3...

ke node 0...

atau ke node 1...

atau ke node 2...

atau ke node 4...



All-Pairs Shortest Path?

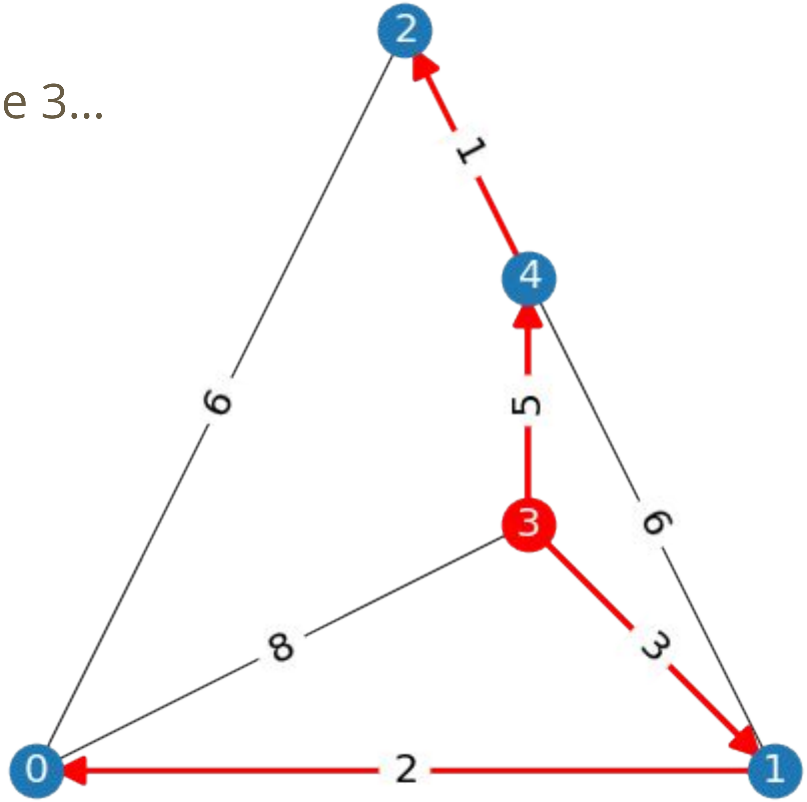
Kita ingin mengetahui *shortest path* dari node 3...

ke node 0...

atau ke node 1...

atau ke node 2...

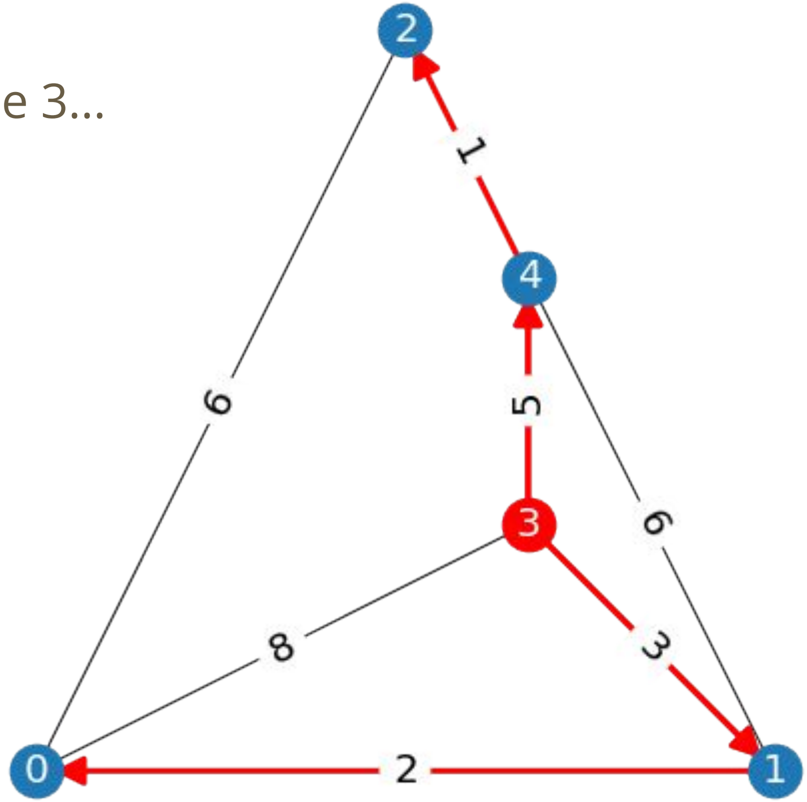
atau ke node 4...



All-Pairs Shortest Path?

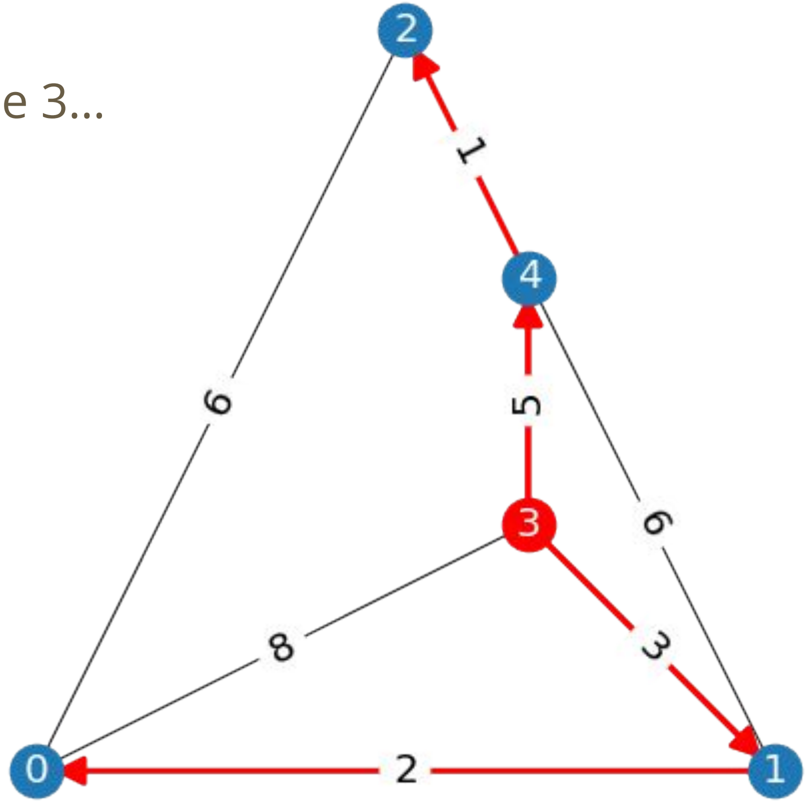
Kita ingin mengetahui *shortest path* dari node 3...

Tujuan	Total *weight*
0	5
1	3
2	6
3	0
4	5



All-Pairs Shortest Path?

Kita ingin mengetahui *shortest path* dari node 3...



Permasalahan ini disebut dengan

Single Source Shortest Path.

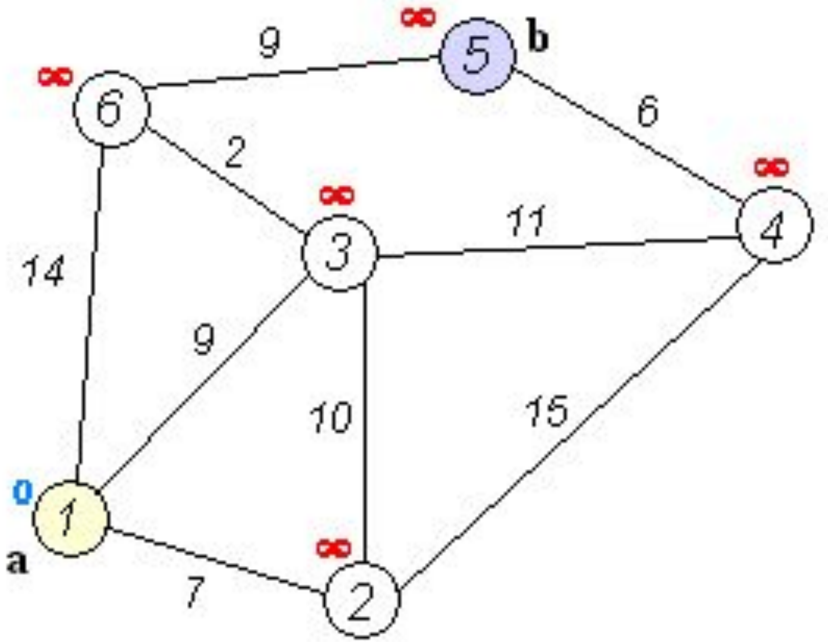
Single Source Shortest Path

Algoritma-algoritma SSSP

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

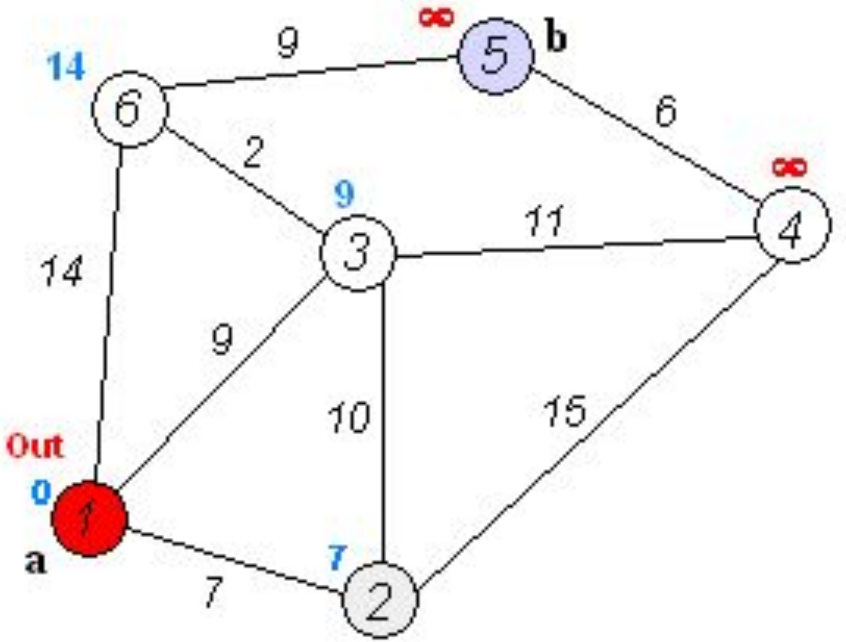
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	∞
3	∞
4	∞
5	∞
6	∞



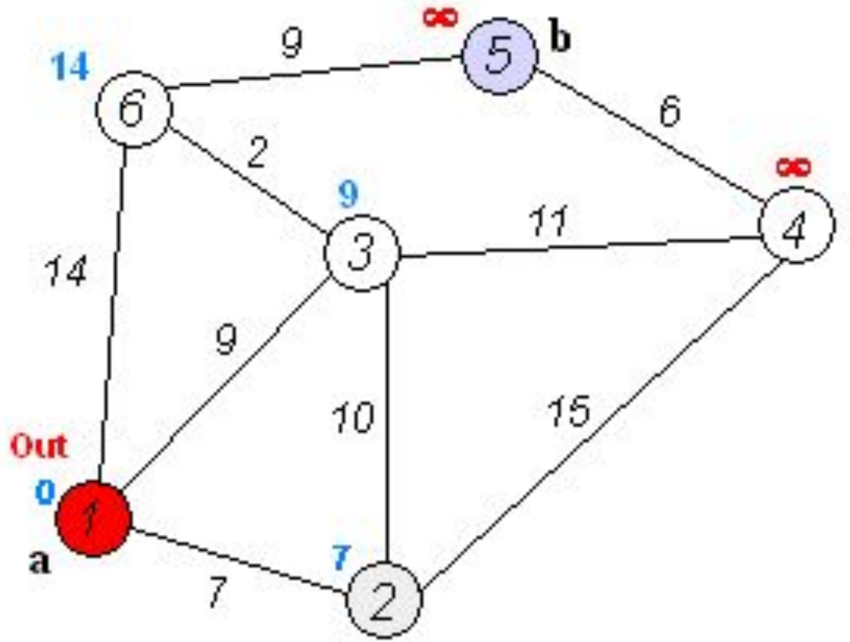
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	∞
5	∞
6	14



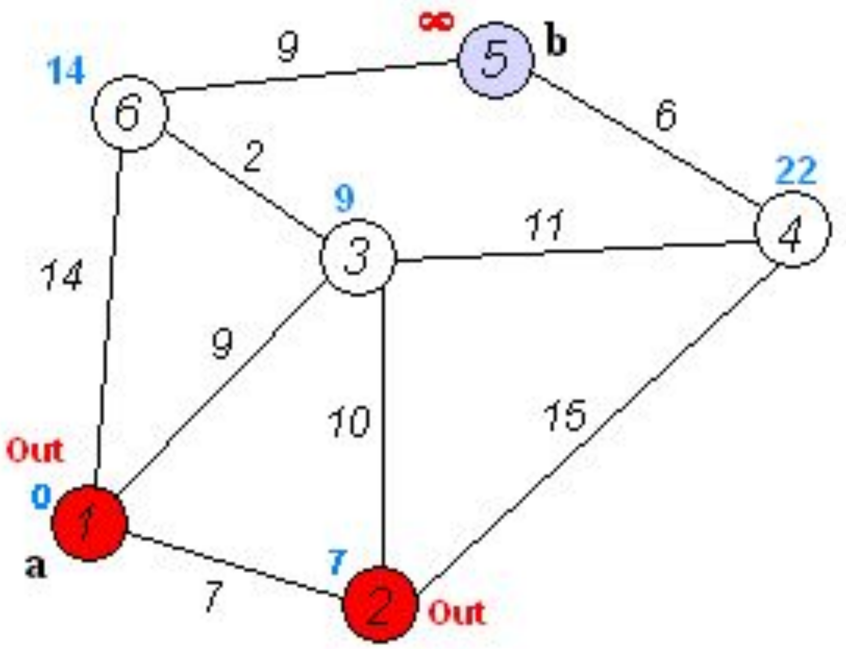
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	∞
5	∞
6	14



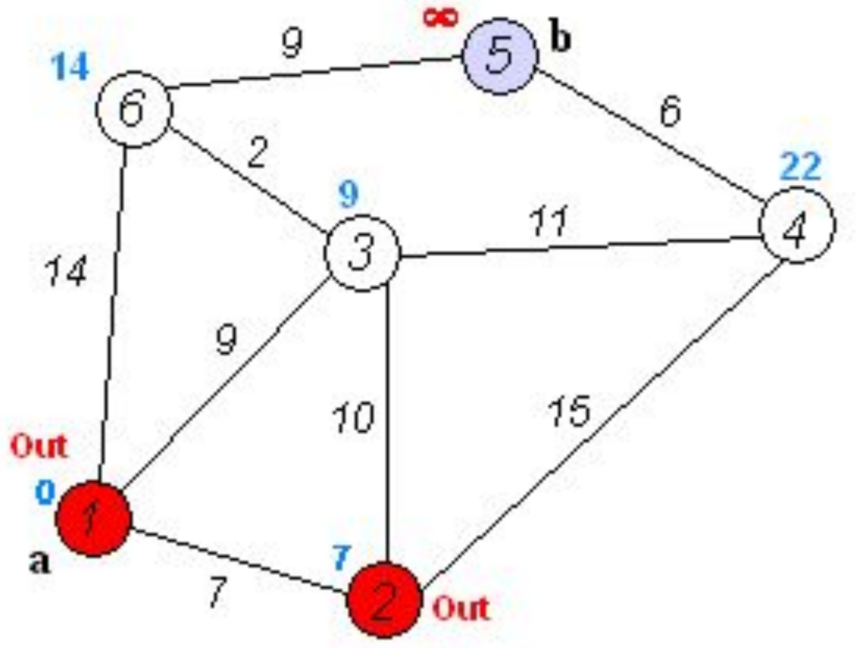
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	22
5	∞
6	14



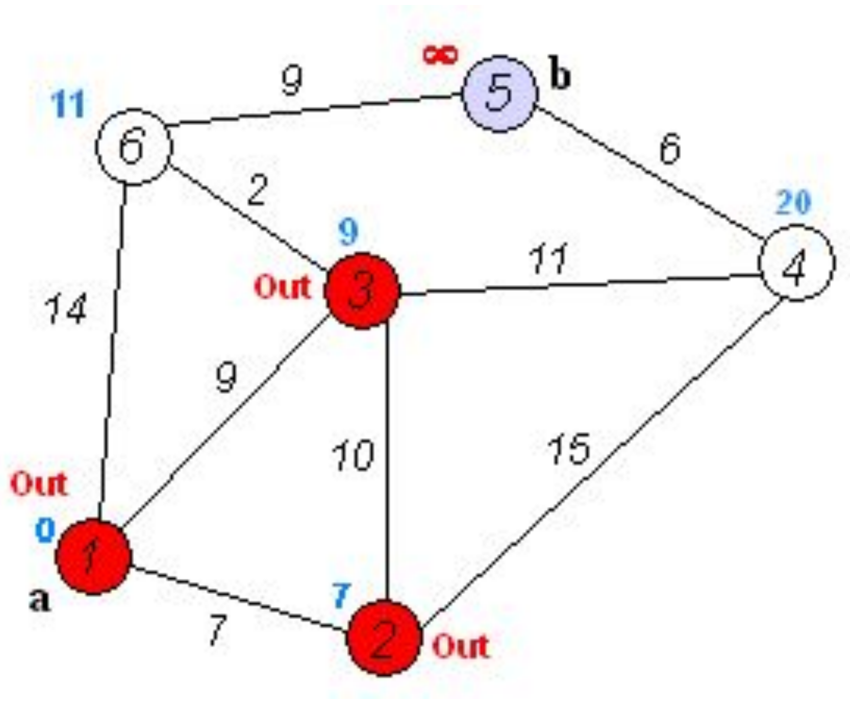
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	22
5	∞
6	14



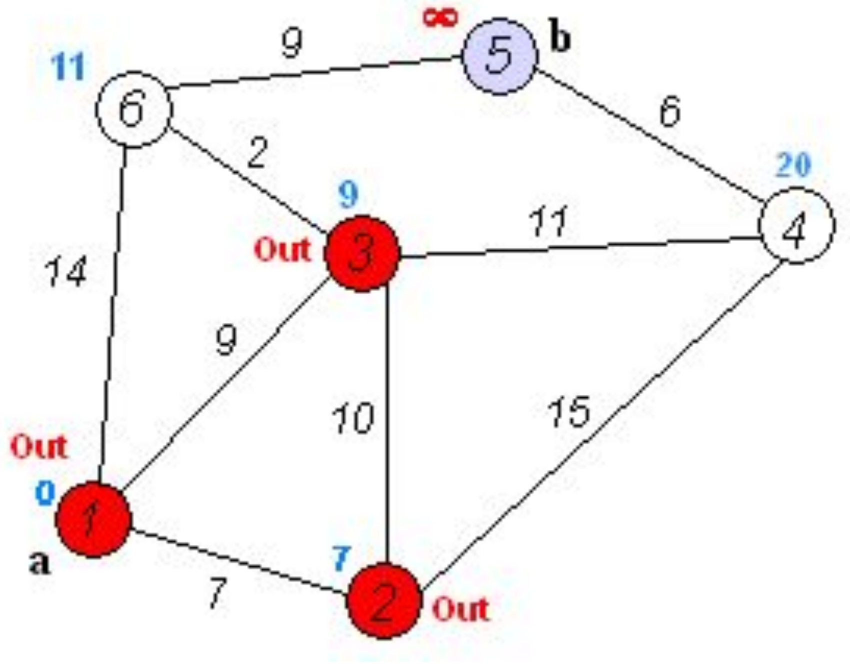
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	20
5	∞
6	11



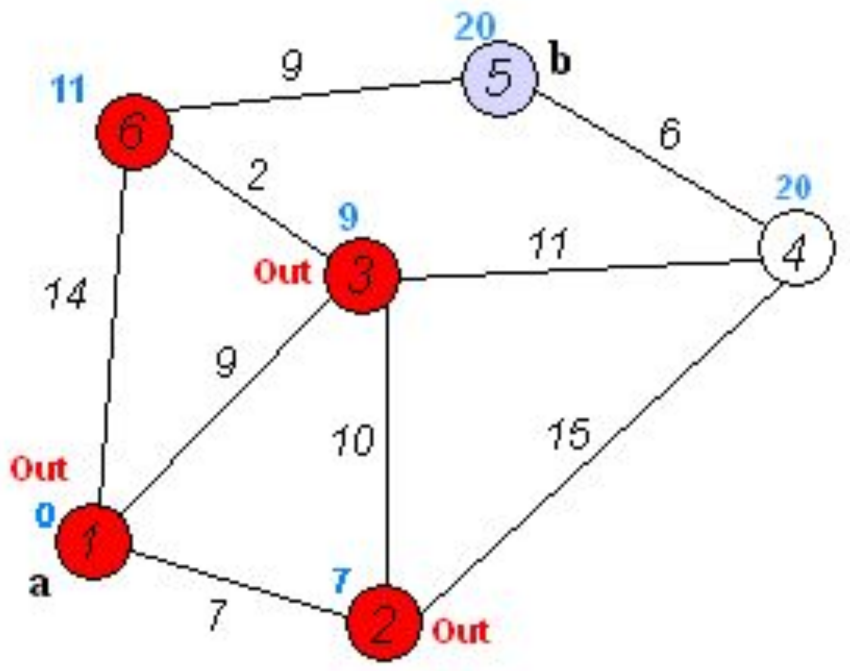
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	20
5	∞
6	11



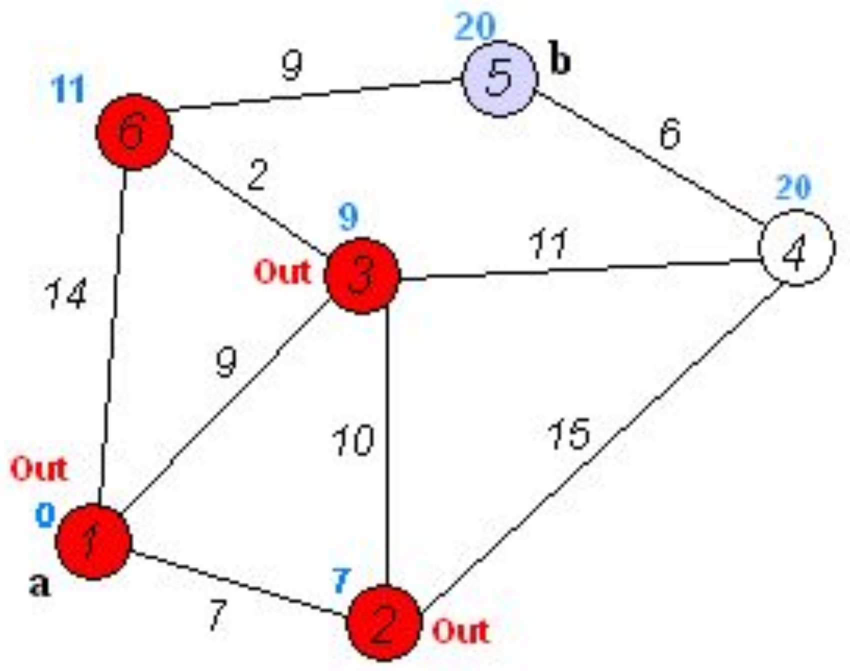
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	20
5	20
6	11



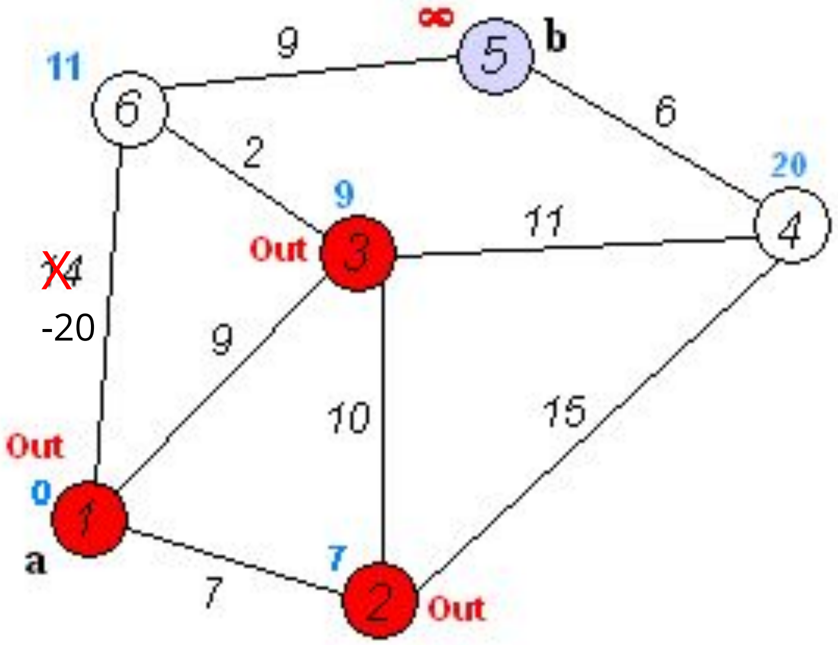
Algoritma Dijkstra

Tujuan	Total *weight*
1	0
2	7
3	9
4	20
5	20
6	11



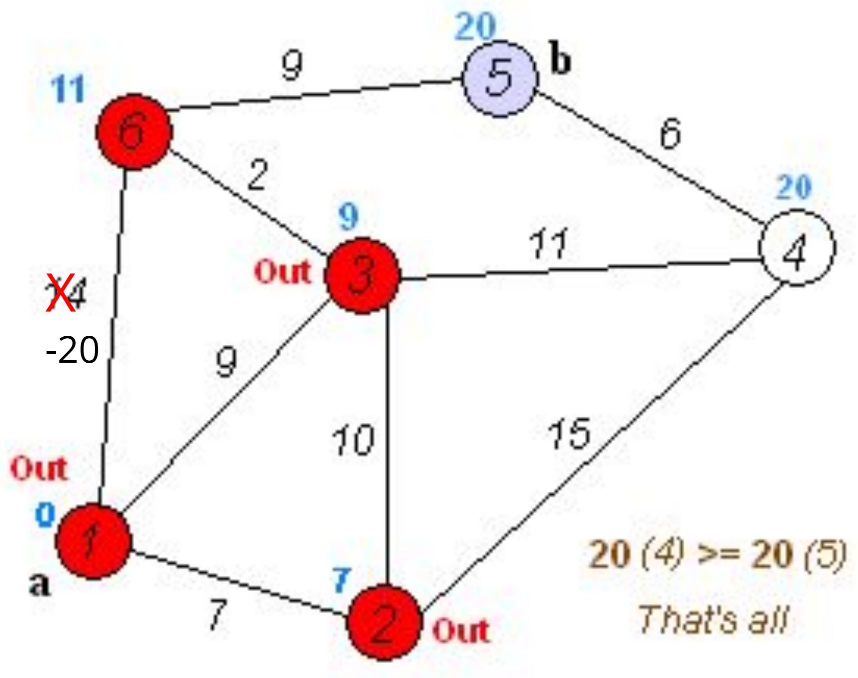
Algoritma Dijkstra -- Permasalahan

Tujuan	Total *weight*
1	0
2	7
3	9
4	20
5	∞
6	11



Algoritma Dijkstra -- Permasalahan

Tujuan	Total *weight*
1	-9
2	7
3	9
4	20
5	20
6	11



Algoritma Dijkstra -- Permasalahan

Negative edge membuat:

Node yang sama dikunjungi berkali-kali,

Edge yang sama dipakai untuk memperbarui jarak berkali-kali,

Jarak yang ditempuh tidak menaik.

Kompleksitas waktu menjadi eksponensial.

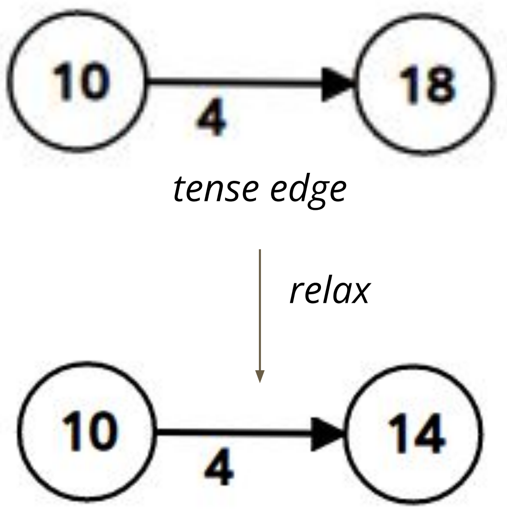
Algoritma Bellman-Ford

Relax semua *tense edge*, ulangi.

Jarak terpendek dari s ke semua *node* lain:

    paling banyak memakai V - 1 *edge*

$\text{dist}(u) + w(u\text{->}v) < \text{dist}(v)$



Algoritma Bellman-Ford

Relax semua *tense edge*, ulangi.

Jarak terpendek dari s ke semua *node* lain:

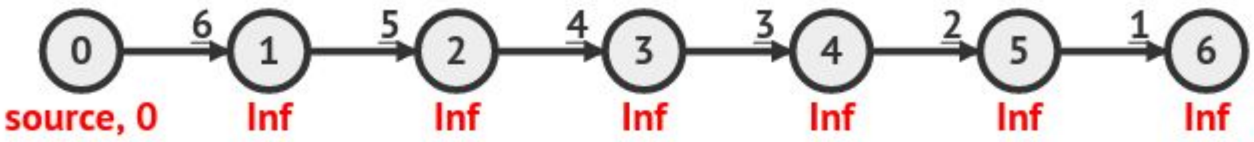
paling banyak memakai V - 1 *edge*

Algoritma 1: Algoritma Bellman-Ford untuk SSSP

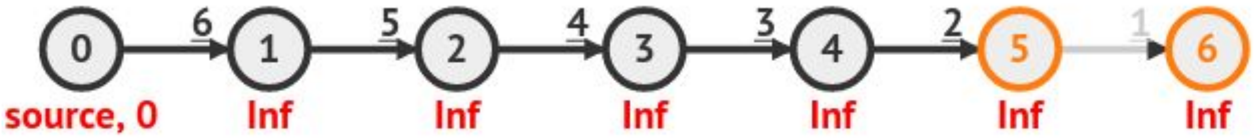
InitSSSP(s)

foreach $l \leftarrow 1$ to $|V| - 1$ **do**
  foreach $u \rightarrow v \in E$ **do**
    if $u \rightarrow v$ *is tense* **then**
      Relax($u \rightarrow v$)
    end
  end
end

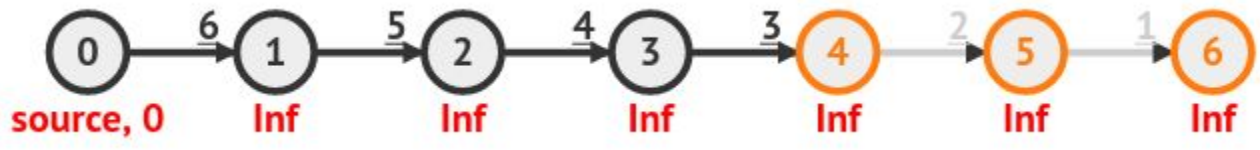
Algoritma Bellman-Ford



Algoritma Bellman-Ford



Algoritma Bellman-Ford



Algoritma Bellman-Ford

Algoritma Bellman-Ford

Algoritma Bellman-Ford

Setelah melakukan *relax* terhadap semua *edge*,

baru 1 *node* yang diketahui jaraknya

Algoritma Bellman-Ford

Setelah melakukan *relax* terhadap semua *edge*,

baru 1 *node* yang diketahui jaraknya

Harus melakukan *relax* semua *edge* sebanyak V - 2 kali lagi.

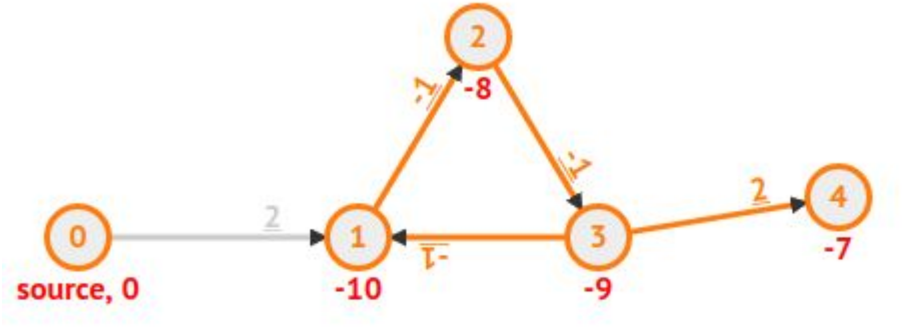
Algoritma Bellman-Ford

Jarak terpendek dari s ke semua *node* lain:

paling banyak menggunakan V - 1 *edge*

Jika setelah *relax* semua *edge* V - 1 kali dan masih terdapat *tense edge:*

negative cycle



Algoritma Bellman-Ford

Algoritma 1: Algoritma Bellman-Ford untuk SSSP

InitSSSP(s)

foreach $l \leftarrow 1$ to $|V| - 1$ **do**
  foreach $u \rightarrow v \in E$ **do**
    if $u \rightarrow v$ *is tense* **then**
      Relax($u \rightarrow v$)
    end
  end
end

foreach $u \rightarrow v \in E$ **do**
  if $u \rightarrow v$ *is tense* **then**
    return "Negative Cycle"
  end
end

All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut...

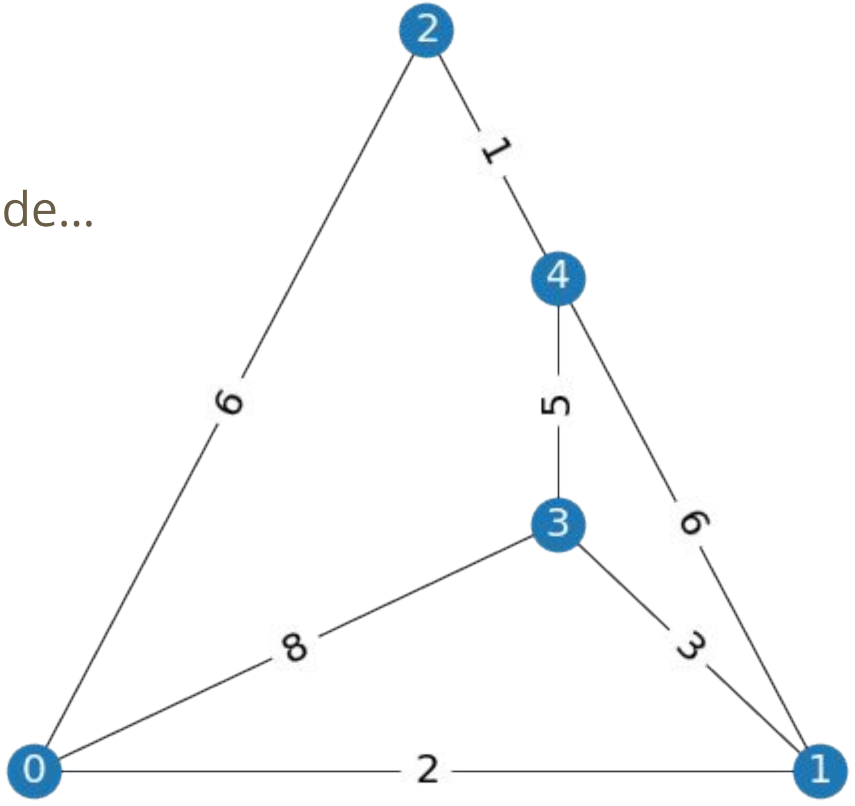
Kita ingin mengetahui *shortest path* dari node...

0 ke mana saja...

atau 1 ke mana saja...

atau 2 ke mana saja...

atau 4 ke mana saja...

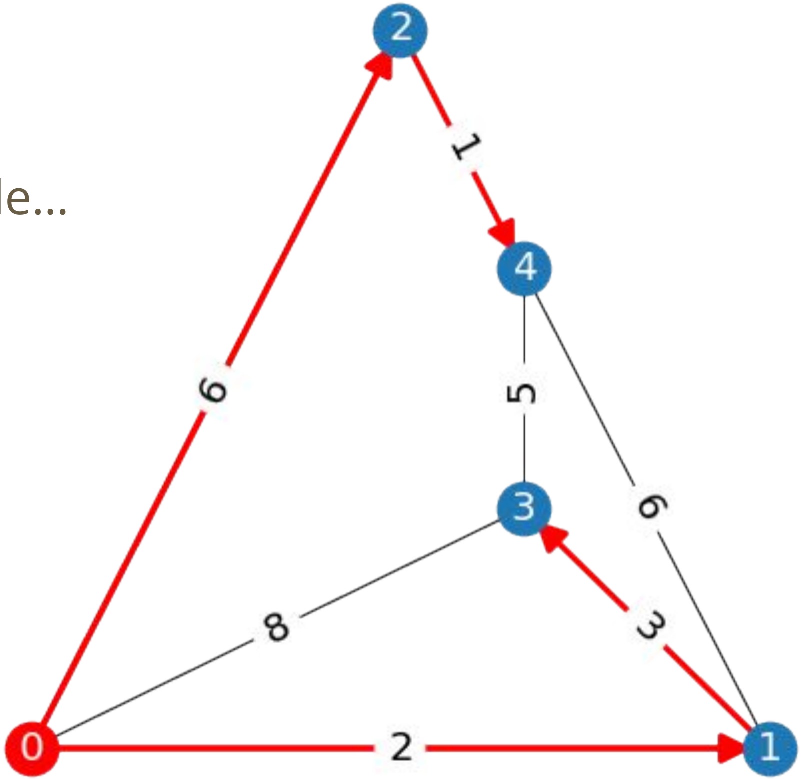


All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut...

Kita ingin mengetahui *shortest path* dari node...

0 ke mana saja...

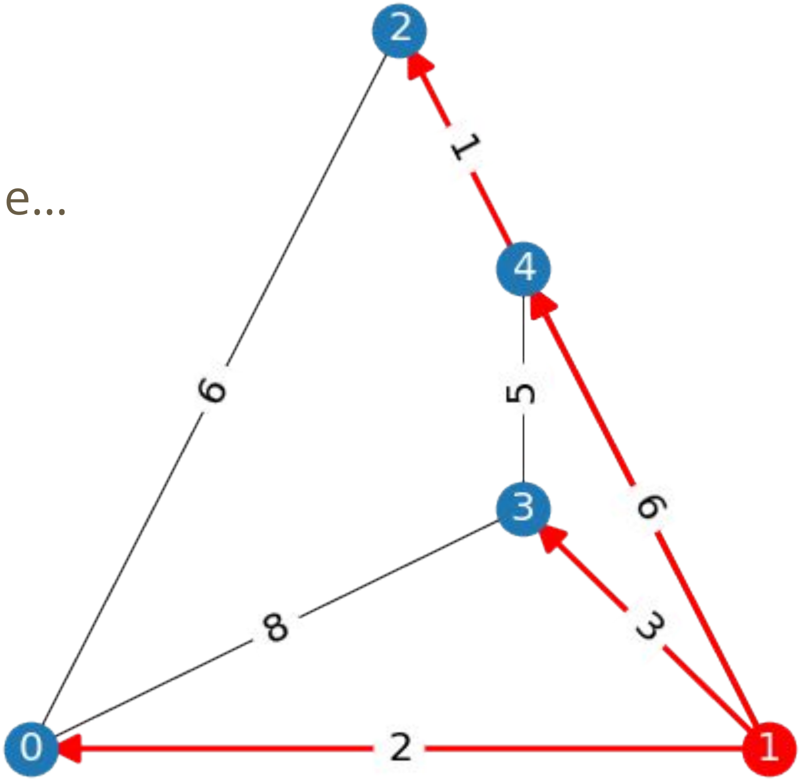


All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut...

Kita ingin mengetahui *shortest path* dari node...

atau 1 ke mana saja...

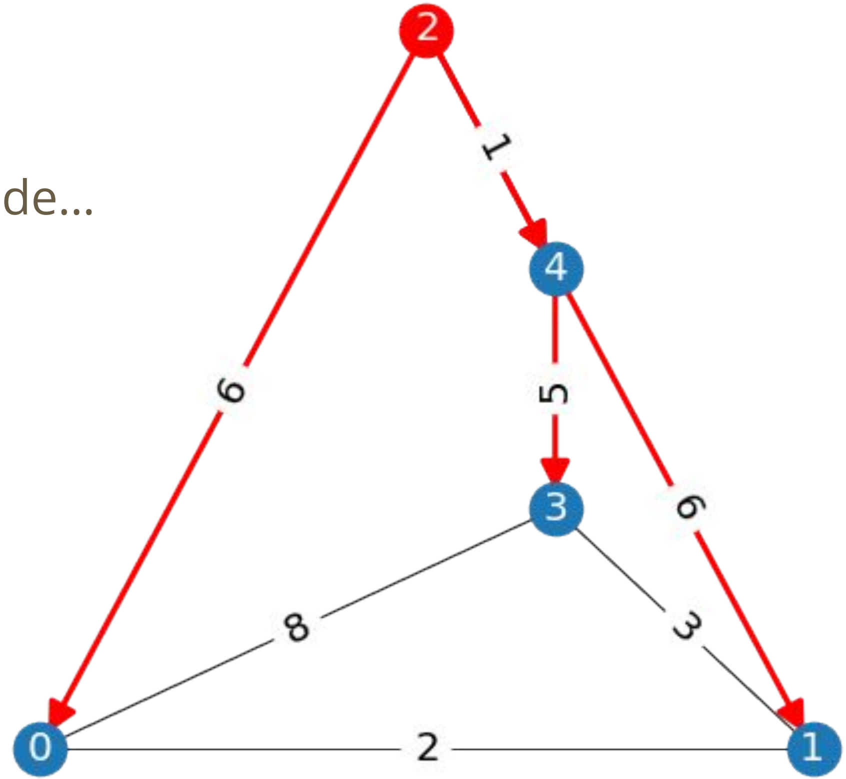


All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut...

Kita ingin mengetahui *shortest path* dari node...

atau 2 ke mana saja...

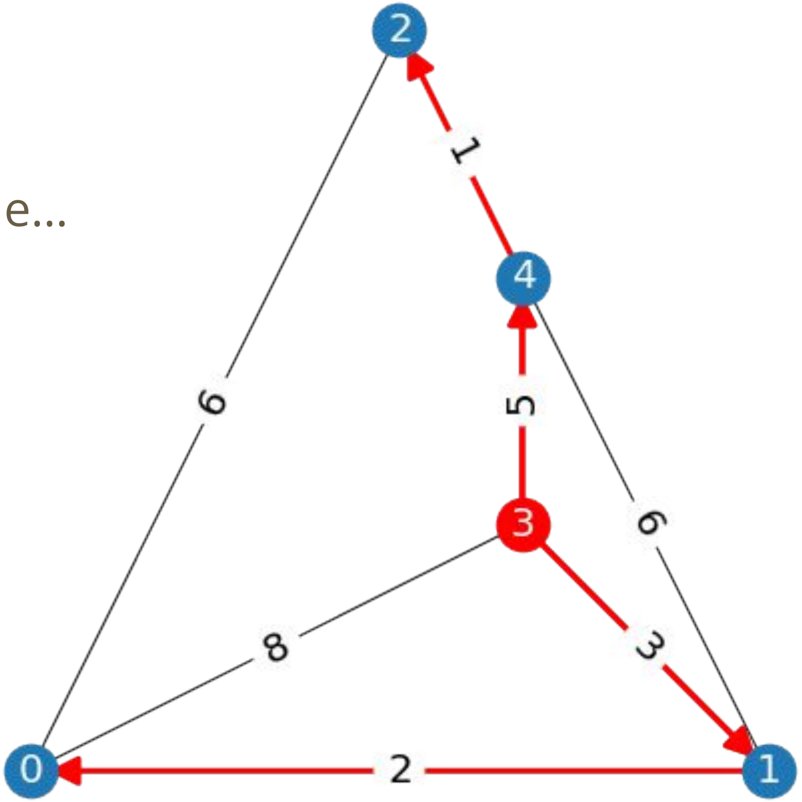


All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut…

Kita ingin mengetahui *shortest path* dari node…

atau 3 ke mana saja…

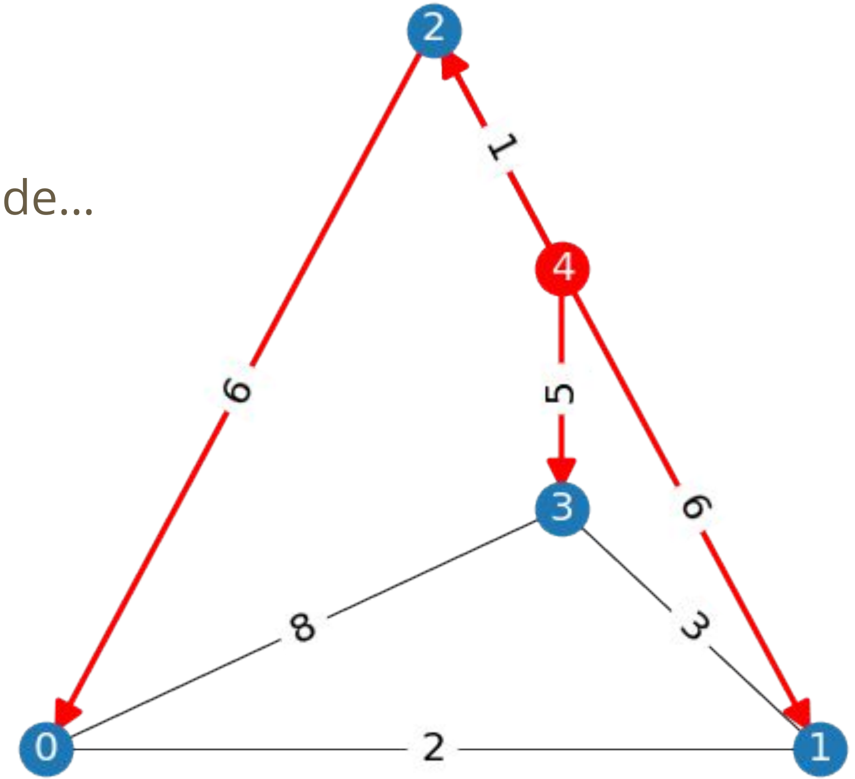


All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut...

Kita ingin mengetahui *shortest path* dari node...

atau 4 ke mana saja...

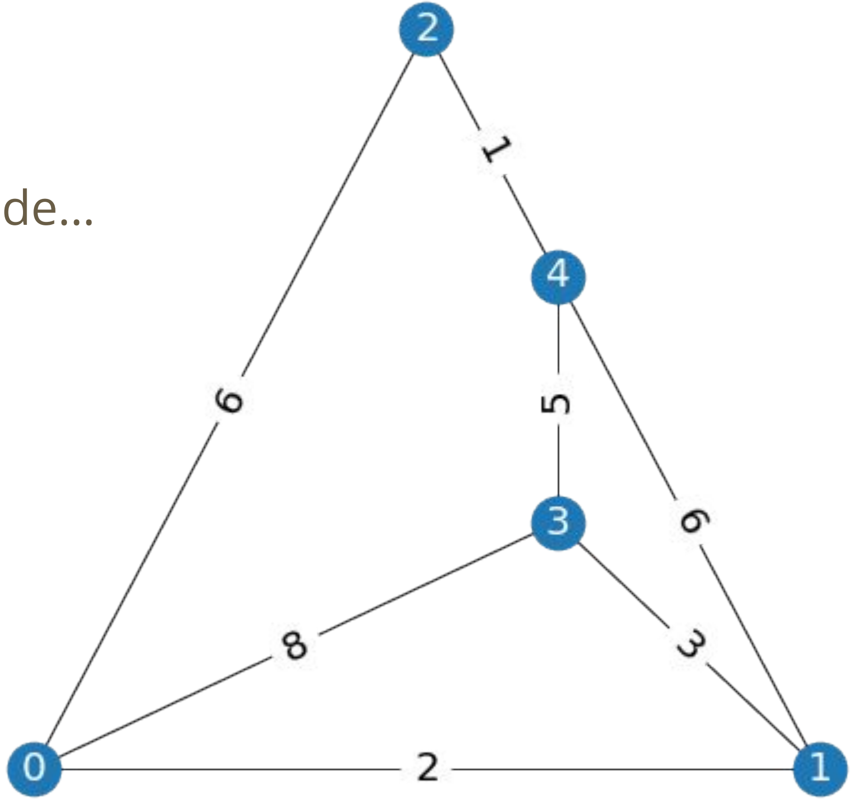


All-Pairs Shortest Path?

Misalkan kita mempunyai graf berikut...

Kita ingin mengetahui *shortest path* dari node...

mana saja ke **mana saja**.



Permasalahan ini disebut dengan

All-Pairs Shortest Path.

Mem-'formal'-kan *All-Pairs Shortest Path* (APSP)

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

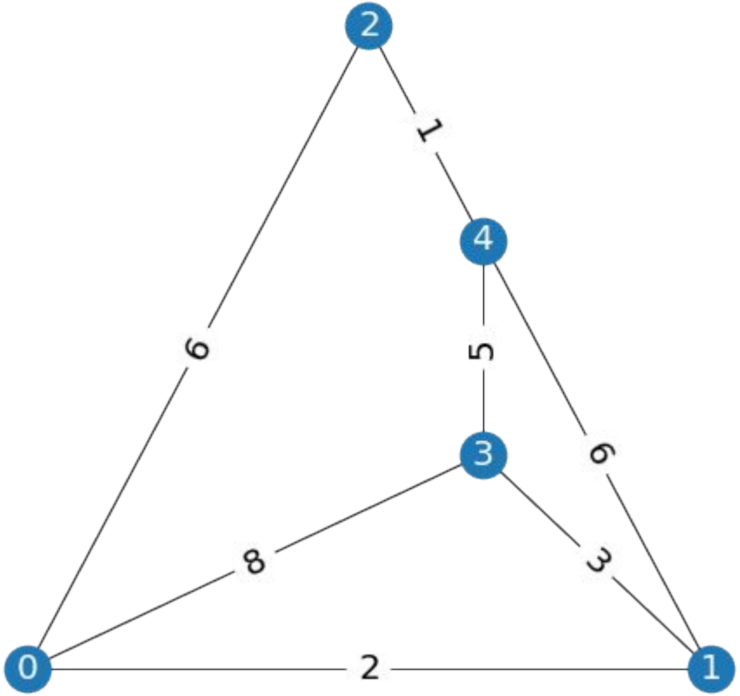
- $dist(u, v)$, yaitu jarak (jumlah *weight* pada *path*) terpendek dari *vertex* u ke v, dan
- $pred(u, v)$, yaitu *vertex* terakhir sebelum v pada *path* dengan jarak terpendek dari *vertex* u ke v.

Mem-'formal'-kan *All-Pairs Shortest Path* (APSP)

Contoh G = (V, E)

		v				
u		0	1	2	3	4
	0	**0**	**2**	**6**	**5**	**7**
	1	**2**	**0**	**7**	**3**	**6**
	2	**6**	**7**	**0**	**6**	**1**
	3	**5**	**3**	**6**	**0**	**5**
	4	**7**	**6**	**1**	**5**	**0**
dist(u,v)						

		v				
u		0	1	2	3	4
	0	**N**	**0**	**0**	**1**	**2**
	1	**1**	**N**	**4**	**1**	**1**
	2	**2**	**4**	**N**	**4**	**1**
	3	**1**	**3**	**4**	**N**	**3**
	4	**2**	**4**	**4**	**4**	**N**
pred(u,v)						



Mem-'formal'-kan *All-Pairs Shortest Path* (APSP)

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 $pred(u, u) = \text{NULL}$.

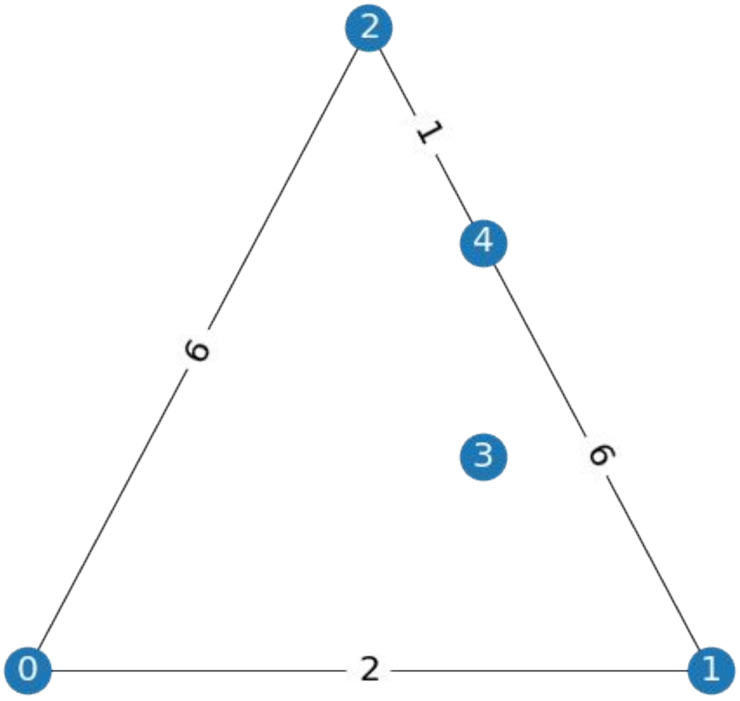
– Apabila tidak ada *path* dari *vertex* u ke v, maka $dist(u,v) = \infty$ dan $pred(u,v) = \text{NULL}$.

dist(u,v)

u \ v	0	1	2	3	4
0	0	2	6	∞	7
1	2	0	7	∞	6
2	6	7	0	∞	1
3	∞	∞	∞	0	∞
4	7	6	1	∞	0

pred(u,v)

u \ v	0	1	2	3	4
0	N	0	0	N	2
1	1	N	4	N	1
2	0	4	N	N	2
3	N	N	N	N	N
4	2	4	4	N	N



– 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 $pred(u, v) = \text{NULL}$.

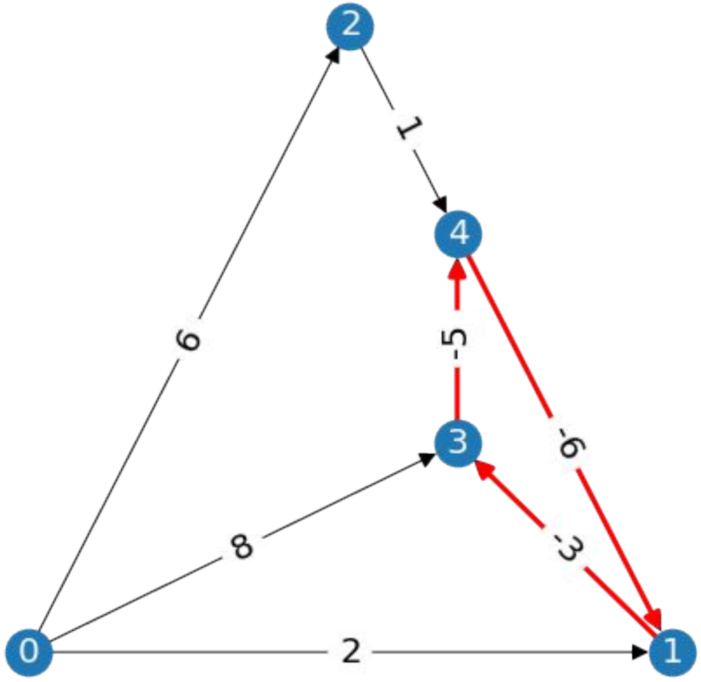
* = – ∞

	v				
u	0	1	2	3	4
0	**0**	*	**6**	*	*
1	∞	*	∞	*	*
2	∞	*	**0**	*	*
3	∞	*	∞	*	*
4	∞	*	∞	*	*

dist(u,v)

	v				
u	0	1	2	3	4
0	**N**	**N**	**0**	**N**	**N**
1	**N**	**N**	**N**	**N**	**N**
2	**N**	**N**	**N**	**N**	**N**
3	**N**	**N**	**N**	**N**	**N**
4	**N**	**N**	**N**	**N**	**N**

pred(u,v)

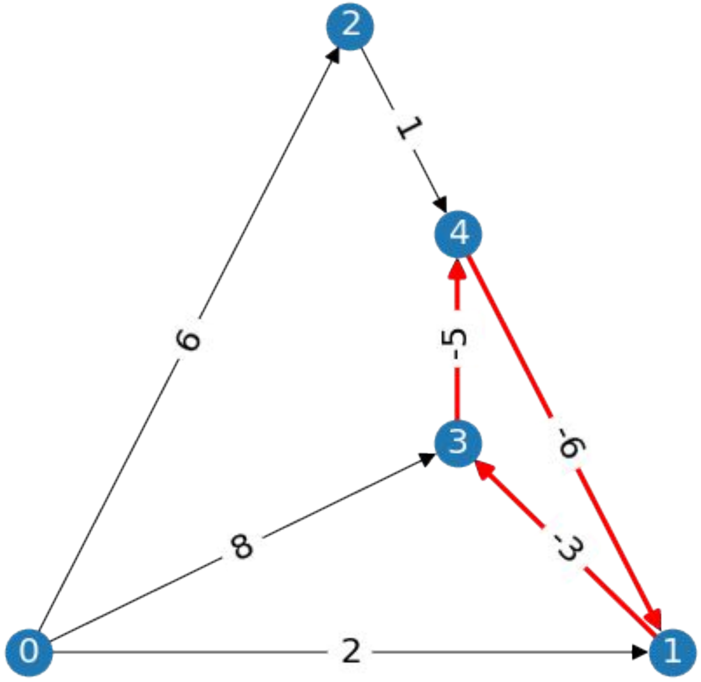


– Apabila tidak terdapat *negative cycle* pada suatu *path* dari *vertex* u ke dirinya sendiri, maka $dist(u, u) = 0$ dan $pred(u, u) = \text{NULL}$.

*** = - ∞**

	v				
u		0	1	2	3	4
	0	**0**	*	**6**	*	*
	1	∞	*	∞	*	*
	2	∞	*	**0**	*	*
	3	∞	*	∞	*	*
	4	∞	*	∞	*	*
dist(u,v)						

	v				
u		0	1	2	3	4
	0	**N**	**N**	**0**	**N**	**N**
	1	**N**	**N**	**N**	**N**	**N**
	2	**N**	**N**	**N**	**N**	**N**
	3	**N**	**N**	**N**	**N**	**N**
	4	**N**	**N**	**N**	**N**	**N**
pred(u,v)						



Menyelesaikan APSP?

Mudah.

SSSP dari setiap *node*.

Seberapa cepat?

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	$O(\lvert V\rvert \times \lvert V\rvert\lvert E\rvert) = O(\lvert V\rvert^4)$

Menyelesaikan APSP dengan SSSP

Table 1. Pendekatan SSSP untuk setiap graf [1]

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

Can we do better?

Menyelesaikan APSP dengan SSSP

Table 1. Pendekatan SSSP untuk setiap graf [1]

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

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).

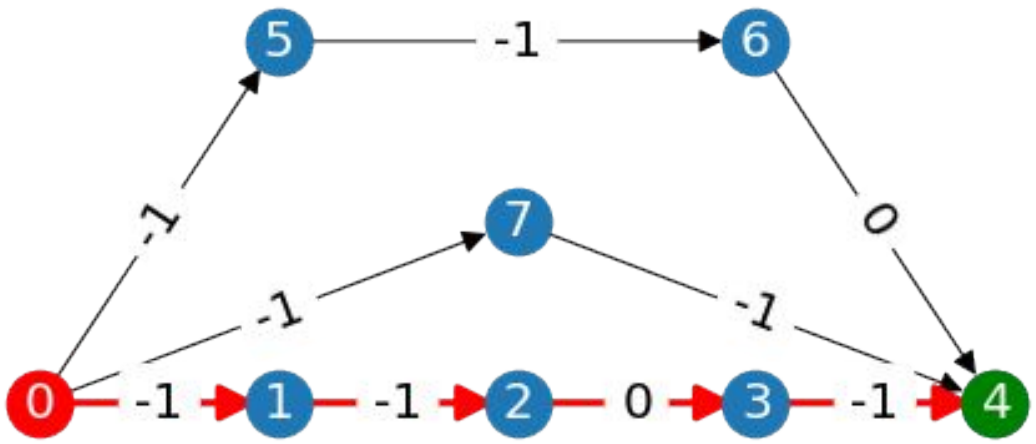
Menyelesaikan APSP dengan SSSP

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).

Apakah kita bisa mengeliminasi ***edge* negatif** pada graf tanpa mengubah *shortest path*-nya?

Reweighting

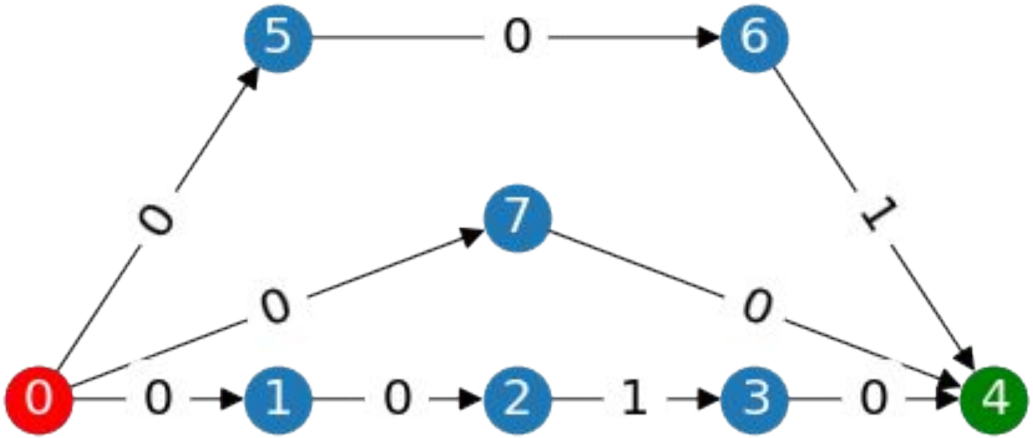
Apakah kita bisa mengeliminasi ***edge* negatif** pada graf tanpa mengubah *shortest path*-nya?



Semua *edge*, weightnya ditambah dengan nilai yang sama agar tidak ada yang negatif?

Reweighting

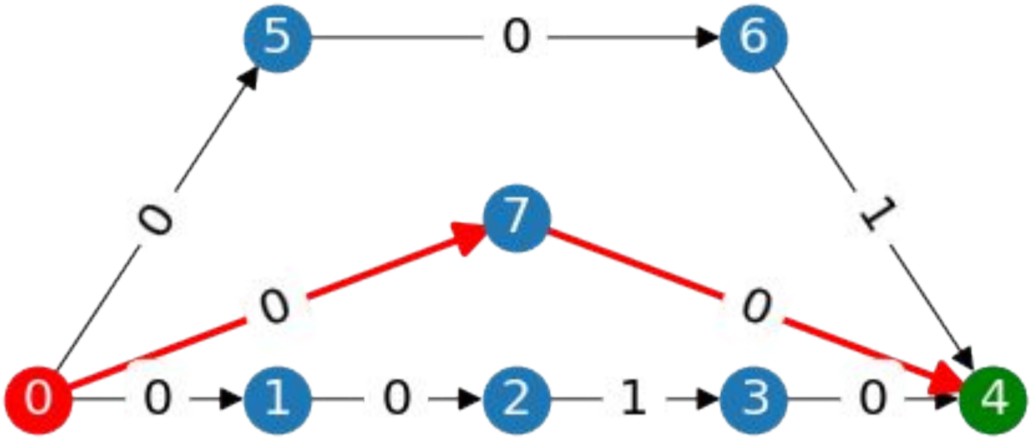
Apakah kita bisa mengeliminasi ***edge* negatif** pada graf tanpa mengubah *shortest path*-nya?



Semua *edge*, weightnya di-tambah agar tidak ada yang negatif?

Reweighting

Apakah kita bisa mengeliminasi ***edge* negatif** pada graf tanpa mengubah *shortest path*-nya?



Tidak bisa.

Reweighting -- Pendekatan Lain

(Donald Johnson, 1973)

Berikan nilai khusus untuk suatu *node* u yaitu π(u).
Definisikan w'(u → v) sebagai *weight* edge yang baru:

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

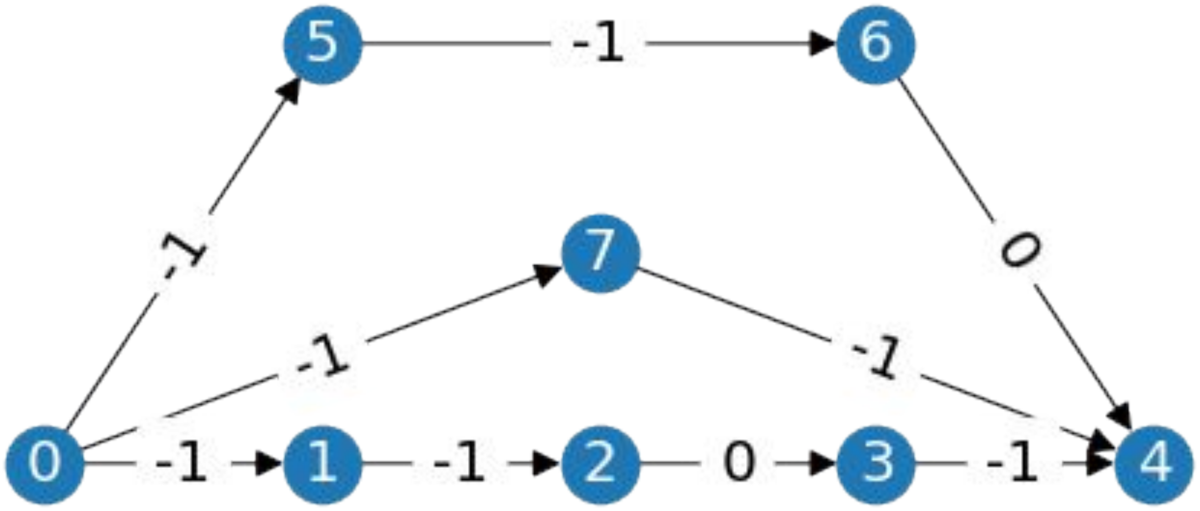
Reweighting -- Pendekatan Lain

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

Perhatikan bahwa rumus baru diatas **tidak akan mengubah *shortest path***.

Reweighting -- Pendekatan Lain

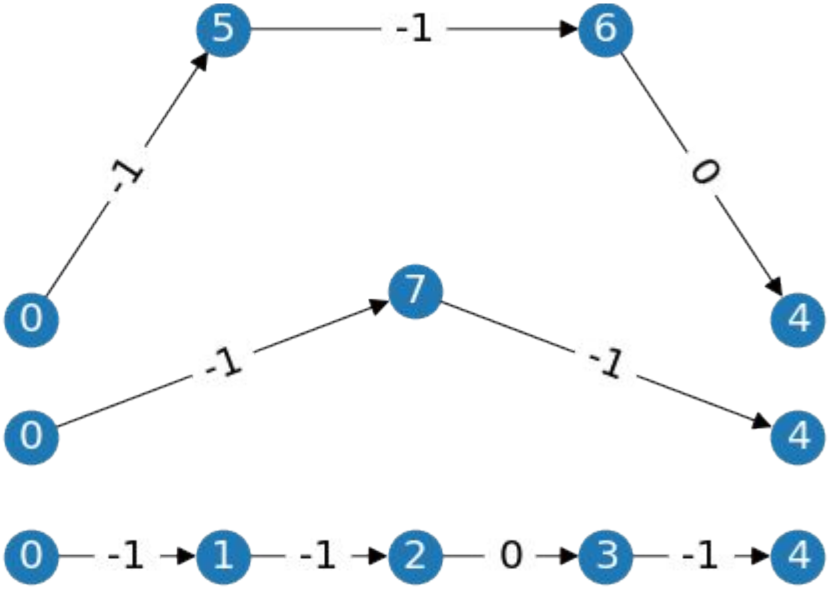
$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$



Reweighting -- Pendekatan Lain

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

Terdapat tiga kemungkinan path dari 0 ke 4.



Reweighting -- Pendekatan Lain

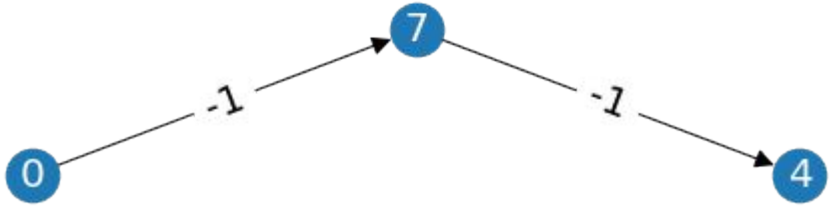
$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$

Total *weight path* **0 ⇒ 4** yang baru:

$$\begin{array}{llll}
w'(0 \to 1) = & \pi(0) & + (-1) & - \pi(1) \\
w'(1 \to 2) = & \pi(1) & + (-1) & - \pi(2) \\
w'(2 \to 3) = & \pi(2) & + 0 & - \pi(3) \\
w'(3 \to 4) = & \pi(3) & + (-1) & - \pi(4) \\
\hline
w'(0 \Rightarrow 4) = & \pi(0) & + (-3) & - \pi(4)
\end{array} +$$

Reweighting -- Pendekatan Lain

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



Total *weight path* **0 ⇒ 4** yang baru:

$w'(0 \to 7) = \quad \pi(0) \quad + (-1) \quad - \pi(7)$

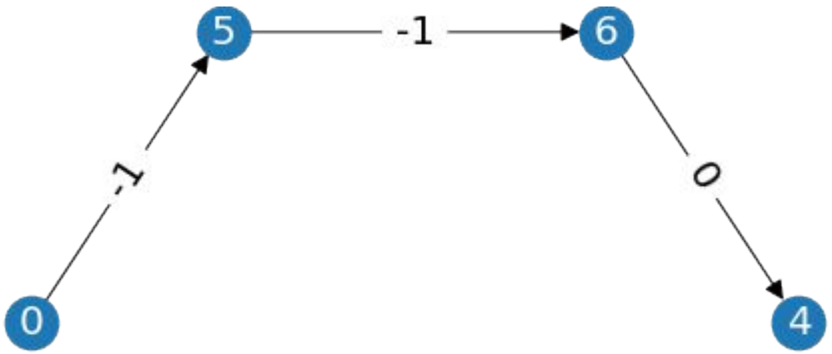
$w'(7 \to 4) = \quad \pi(7) \quad + (-1) \quad - \pi(4)$

\-- +

$w'(0 \Rightarrow 4) = \quad \pi(0) \quad + (-2) \quad - \pi(4)$

Reweighting -- Pendekatan Lain

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



Total *weight path* **0 ⇒ 4** yang baru:

w'(0 ⟶ 5) =	π(0)	+ (-1)	- π(5)
w'(5 ⟶ 6) =	π(5)	+ (-1)	- π(6)
w'(6 ⟶ 4) =	π(6)	+ 0	- π(4)

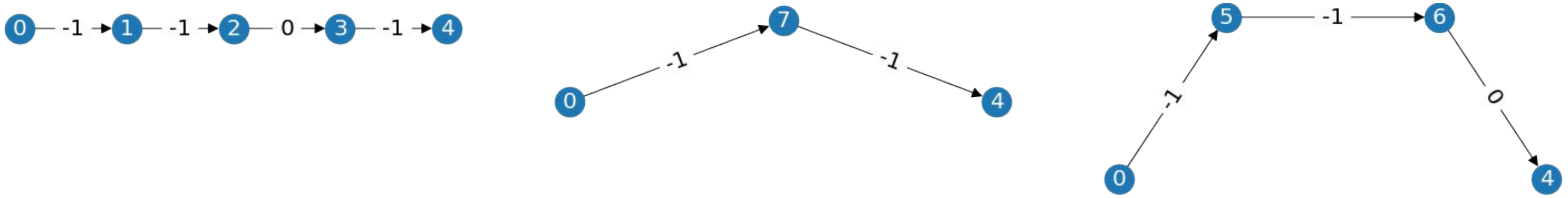
-- +

w'(0 ⇒ 4) =	π(0)	+ (-2)	- π(4)

Reweighting -- Pendekatan Lain

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

Membandingkan ketiga total *weight*...



π(0) + (-3) **- π(4)**

π(0) + (-2) **- π(4)**

π(0) + (-2) **- π(4)**

Perhatikan bahwa *total weight* yang baru untuk masing-masing *path* berubah sebanyak nilai yang sama. Maka, *shortest path* tidak akan berubah, yaitu **yang paling kiri**.

Reweighting -- Pendekatan Lain

Secara umum, untuk setiap *path* u ⇒ v, total *weight* yang baru adalah sebagai berikut:

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

Maka, *shortest path* tidak akan berubah apabila menggunakan rumus baru tersebut.

Algoritma Johnson -- Mengatur nilai π

Pertanyaan berikutnya: Bagaimana caranya kita memberikan nilai π ke setiap *node* pada suatu graf, agar *edge* negatifnya hilang?

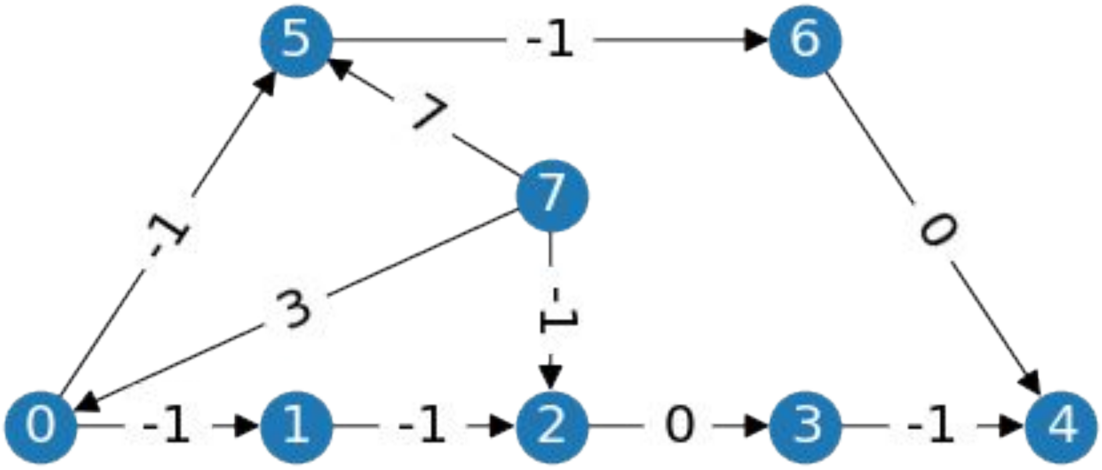
Algoritma Johnson -- Mengatur nilai π

Dengan definisi tersebut, kita tinggal menentukan nilai π 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 $s \Rightarrow v$ sehingga *weight* yang baru adalah:

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

Algoritma Johnson -- Mengatur nilai π

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

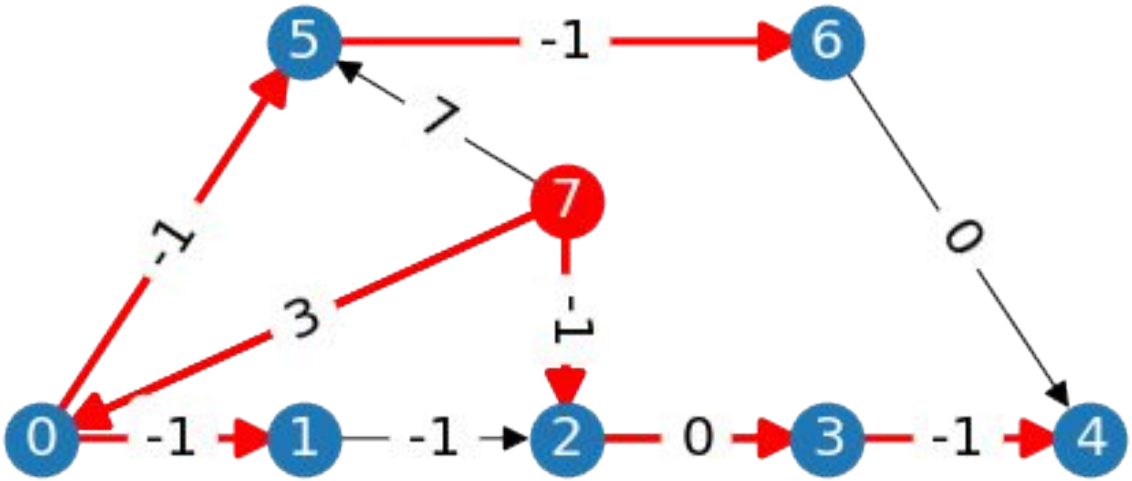


Perhatikan *node* 7 bisa mencapai semua *node* lain.

Untuk contoh ini, kita ambil
s = 7

Algoritma Johnson -- Mengatur nilai π

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

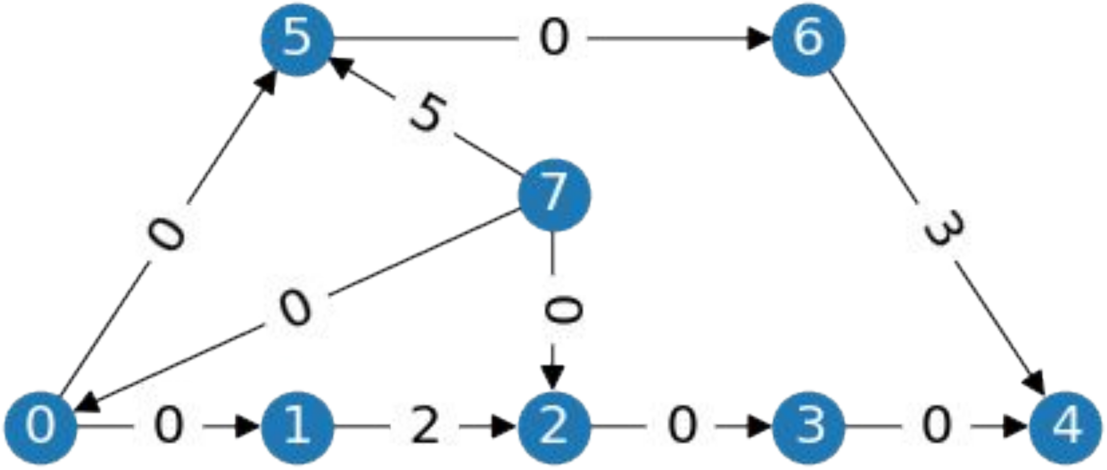


Kemudian, kita ubah *weight*-nya.

u	dist(7,u)
0	3
1	2
2	-1
3	-1
4	-2
5	2
6	1
7	0

Algoritma Johnson -- Mengatur nilai π

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



Tidak ada *edge* negatif lagi!

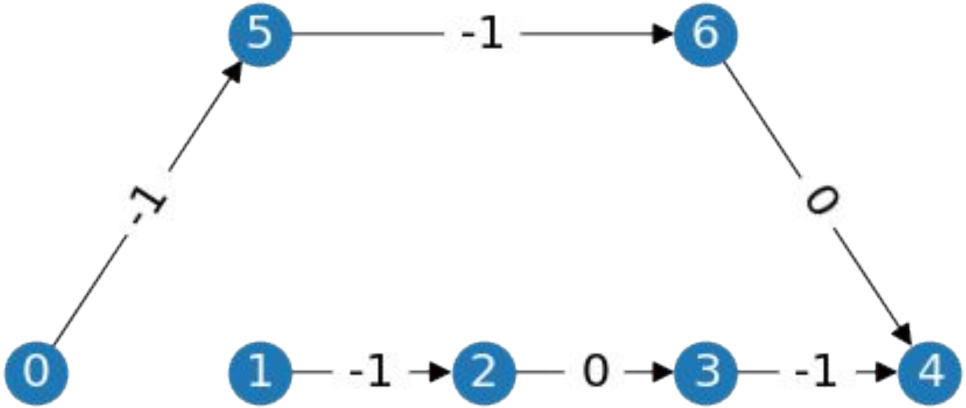
Algoritma Johnson -- Mengatur nilai π

Bisa dibuktikan (secara matematis) bahwa metode tersebut akan selalu menghasilkan *weight edge* yang baru yang nonnegatif.

Bagaimana jika tidak ada *node* s yang memenuhi?

Algoritma Johnson -- Mengatur nilai π

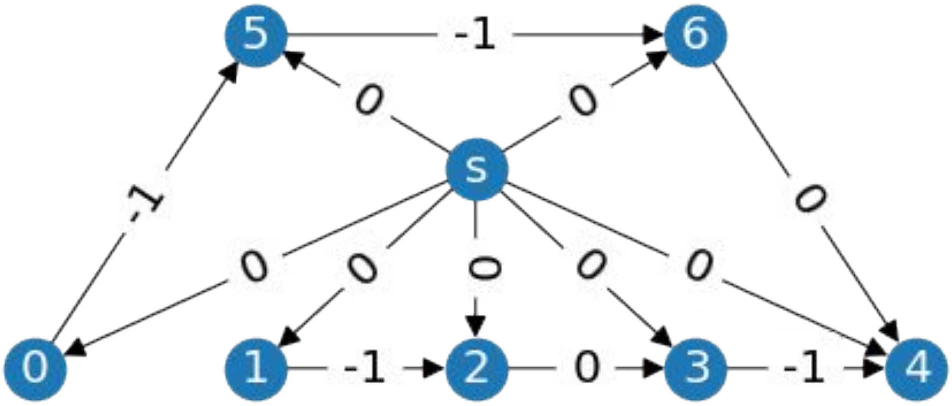
$$w'(u \to v) = dist(s,u) + w(u \to v) - dist(s,v)$$



Bagaimana jika tidak ada *node* s yang memenuhi?

Algoritma Johnson -- Mengatur nilai π

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

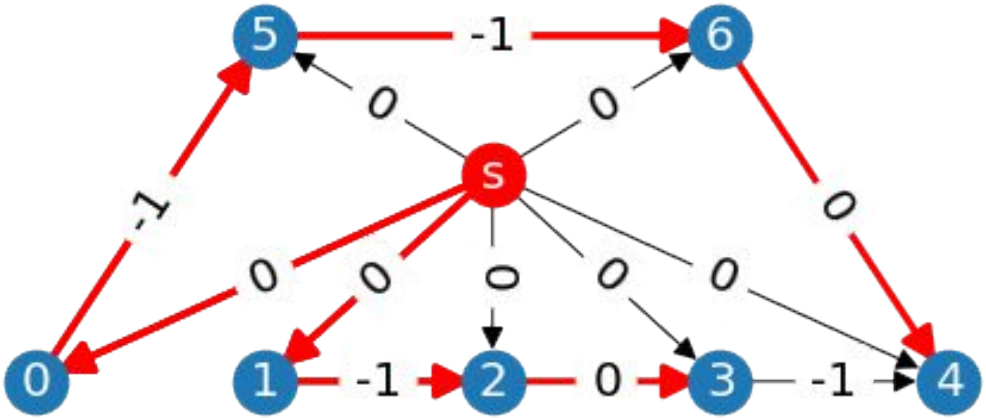


Bagaimana jika tidak ada *node* s yang memenuhi?

Kita buat *node* s baru, dan tambahkan *edge* ke setiap *node* lain dengan *weight* 0. (Pastikan *node* lain tidak bisa mencapai s!)

Algoritma Johnson -- Mengatur nilai π

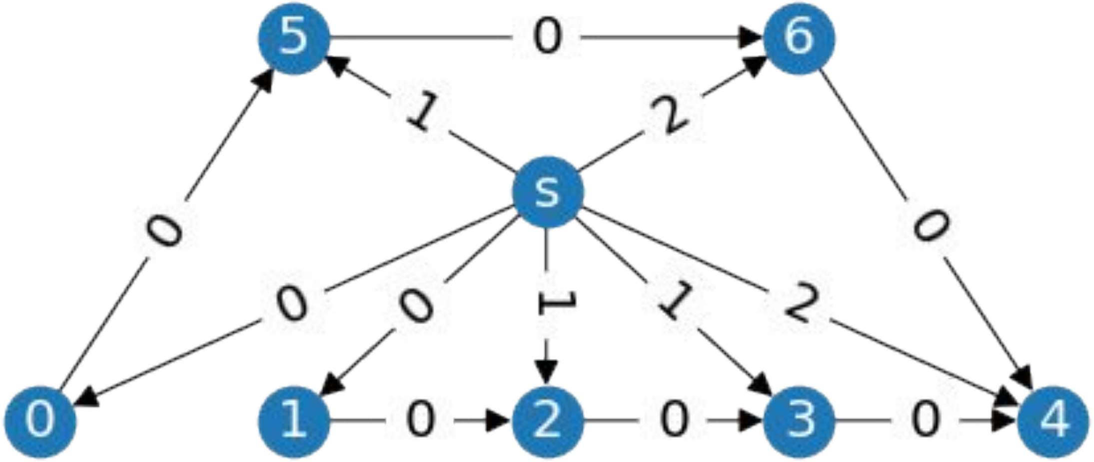
$$w'(u \to v) = dist(s,u) + w(u \to v) - dist(s,v)$$



u	dist(s,u)
0	0
1	0
2	-1
3	-1
4	-2
5	-1
6	-2
s	0

Algoritma Johnson -- Mengatur nilai π

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



Tidak ada *edge* negatif lagi!

Algoritma Johnson -- Langkah berikutnya

Apa yang bisa kita lakukan dengan graf tanpa *edge* negatif ini?

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	$O(\lvert V\rvert \times \lvert V\rvert \lvert E\rvert) = O(\lvert V\rvert^4)$

Kita bisa gunakan **algoritma Dijkstra** untuk mendapatkan APSP-nya!

Algoritma Johnson -- Langkah

- Cari *node* s
  - **Agar tidak pusing, buat *node* s dan tambahkan *edge* ke *node* lain dengan weight 0**
- Hitung dist(s, u) ke setiap *node* lain u
  - Bisa menggunakan Bellman-Ford
- *Update weight* setiap *edge* dengan yang baru
  - Semua *edge* akan menjadi nonnegatif!
- Untuk setiap *node*, gunakan Dijkstra untuk mendapatkan APSP-nya!
  - Lebih cepat dari Bellman-Ford!

Algoritma Johnson -- Poin Penting

- Sayangnya, algoritma ini tidak bisa digunakan untuk graf dengan *negative cycle*.
  - Kita bisa cek apakah graf mempunyai *negative cycle* saat menggunakan **Bellman-Ford**.

- Total jarak pada hasil APSP bukanlah total jarak yang sebenarnya!
  - Karena *weight edge*-nya berubah
  - Bisa diperbaiki

Algoritma Johnson -- Poin Penting

- Total jarak pada hasil APSP bukanlah total jarak yang sebenarnya!
  - Karena *weight edge*-nya berubah
  - Bisa diperbaiki

Perhatikan bahwa untuk suatu path u ⇒ v:

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

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

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 $v \in V$ **do**

  add a new edge $s \to v$ with $w(s \to v) = 0$

end

dist[s][$\cdot$] $\leftarrow$ Bellman-Ford(G, s, w)

if Bellman-Ford finds a *negative cycle* **then**

  stop (fail)

end

foreach $u \to v \in E$ **do**

  $w'(u \Rightarrow v) \leftarrow dist[s][u] + w(u \to v) - dist[s][v]$

end

foreach $u \in v$ **do**

  $dist'[u][\cdot] \leftarrow \text{Dijkstra}(G, u, w')$

end

foreach $u \in V$ **do**

  foreach $v \in V$ **do**

    $dist[u][v] = dist'[u][v] - dist[s][u] + dist[s][v]$

  end

end

return *dist* without s

Algoritma Johnson -- Kompleksitas Waktu

- Membuat *node* s dan *edge* ke setiap *node* lain
  - → **O(|V|)**
- Hitung dist(s, u) menggunakan Bellman-Ford
  - → **O(|E||V|) = O(|V|^3)**
- *Update weight* setiap *edge* dengan yang baru
  - → **O(|E|)**
- Untuk setiap *node*, gunakan Dijkstra untuk mendapatkan APSP
  - → **O(|V| x |E| log |V|) = O(|V|^3 log |V|)**
- Perbaiki total panjang APSP
  - → **O(|V|^2)**

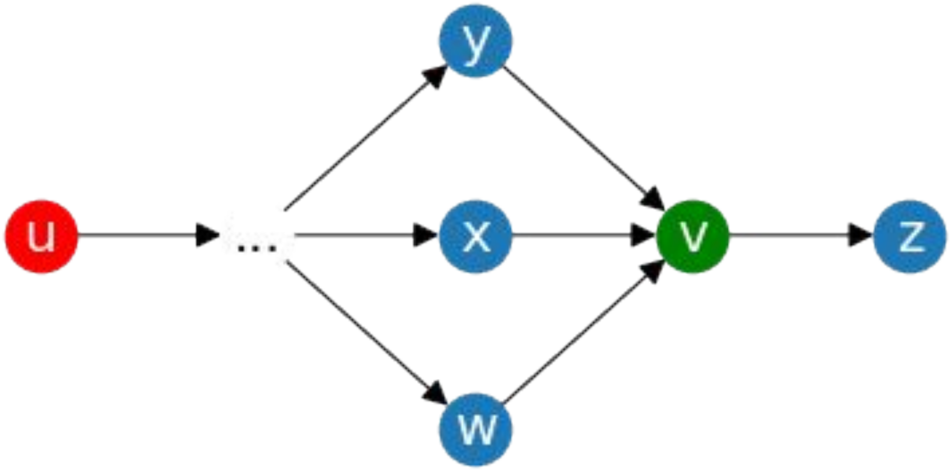
Kompleksitas waktu total = **O(|V| x |E| log |V|) = O(|V|^3 log |V|)**

Akhir dari algoritma Johnson.

*src: https://www.pinterest.com/pin/405535141443781694/

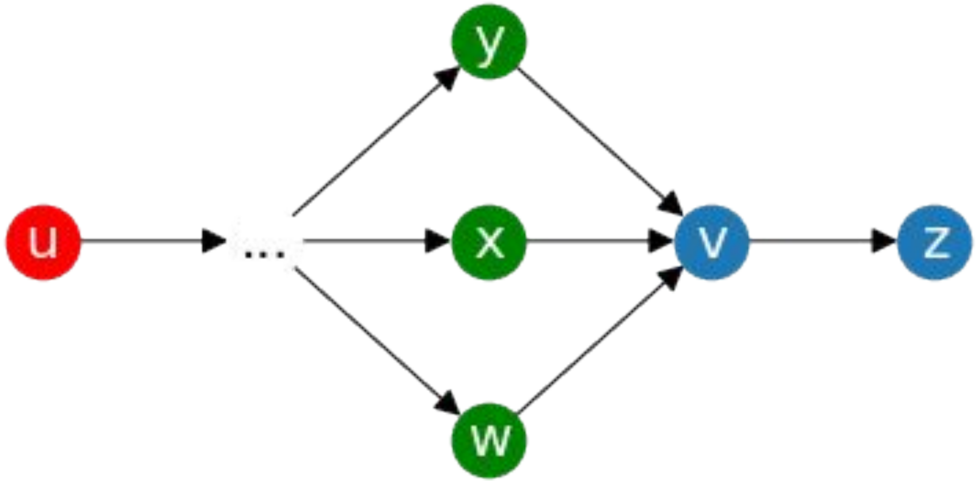
Pendekatan Rekursif -- Berpikir Secara Rekursif

Misalkan kita ingin mengetahui panjang *shortest path* dari u ke v...



Pendekatan Rekursif -- Berpikir Secara Rekursif

Cari panjang *shortest path* dari u ke setiap *node* yang bisa mencapai v dan kemudian bandingkan mana yang paling pendek dalam mencapai v.



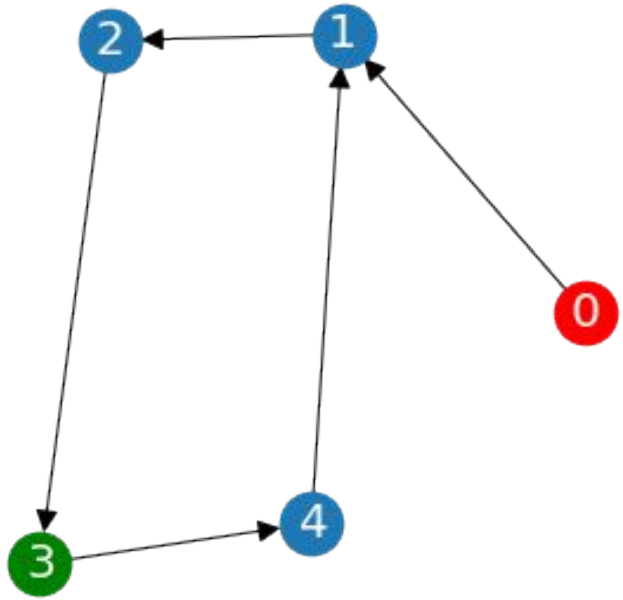
Pendekatan Rekursif -- Berpikir Secara Rekursif

Secara matematis...

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

Pendekatan Rekursif -- Berpikir Secara Rekursif

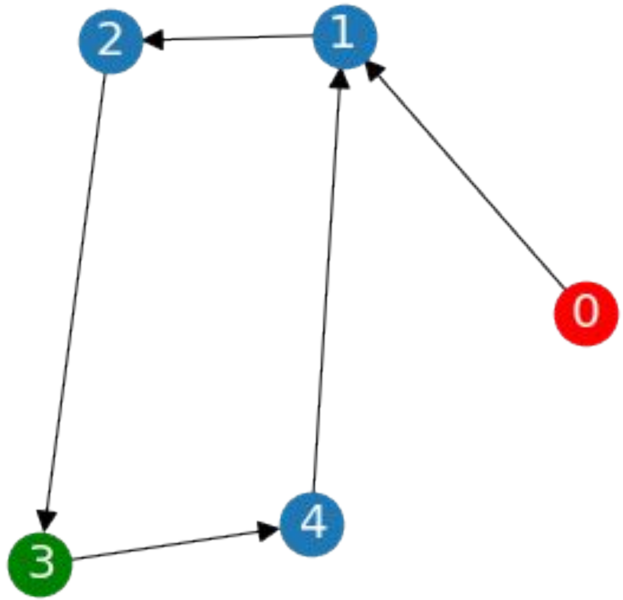
Sayangnya, metode rekursif ‘biasa’ tidak bisa dilakukan untuk graf dengan *cycle*.



Pendekatan Rekursif -- Berpikir Secara Rekursif

Sayangnya, metode rekursif ‘biasa’ tidak bisa dilakukan untuk graf dengan *cycle*.

dist(0,3) = dist(0,2) + w(2 → 3)
dist(0,2) = dist(0,1) + w(1 → 2)
dist(0,1) = min(dist(0,0) + w(0 → 1)
dist(0,4) + w(4 → 1))
dist(0,4) = **dist(0,3)** + w(3 → 4)
...



Pendekatan Rekursif -- Berpikir Secara Rekursif

Observasi Bellman-Ford:
Pada suatu graf ***tanpa negative cycle****, shortest path u ⇒ v pasti paling banyak menggunakan* (|V|-1) *edge*.

Kita bisa memberikan ‘batasan’ pada rekursi sehingga rekurens menjadi berikut:

dist(u,v,l) = Panjang *shortest path* u ⇒ v dengan **menggunakan paling banyak** ***l edge***.

Pendekatan Rekursif -- Berpikir Secara Rekursif

dist(u,v,l) = Panjang *shortest path* u ⇒ v dengan menggunakan paling banyak *l* *edge*.

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

Perhatikan bahwa *dist(u,v)* = *dist(u, v, |V| - 1)*

Pendekatan Rekursif -- *Dynamic Programming*

Rekurens tersebut bisa dihitung dengan *dynamic programming*.

- Mulai dari *base case* l = 0, lalu
- Isi tabel *memo* dari l = 1, 2, 3, sampai |V| - 1

Algoritma tersebut dirancang pertama kali oleh Alfonso Shimbel pada tahun 1954.

Algoritma 2: Algoritma Shimbel

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \rightarrow v)$
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 $u \in V$ **do**
  foreach $v \in V$ **do**
    if $u = v$ **then** $dist[u][v][0] \leftarrow 0$
    else $dist[u][v][0] \leftarrow \infty$
  end
end

foreach $l \leftarrow 1$ to $|V| - 1$ **do**
  foreach $u \in V$ **do**
    foreach $v \in V, v \neq u$ **do**
      $dist[u][v][l] \leftarrow dist[u][v][l-1]$
      foreach $(x \rightarrow v) \in E$ **do**
        $dist[u][v][l] \leftarrow \min(dist[u][v][l], dist[u][x][l-1] + w(x \rightarrow v))$
      end
    end
  end
end

return $dist$

Dynamic Programming -- Kompleksitas Waktu

- *For loop* pertama untuk mengisi *base case*
  - → **O(|V|^2)**
- *For loop* kedua untuk mengisi tabel keseluruhan
  - Asumsi *edge* paling banyak yang keluar suatu *node* adalah |V|
  - → **O(|V| x |V| x |V| x |V|) = O(|V|^4)**

Kompleksitas waktu total = **O(|V|^4)**

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
```

Dynamic Programming -- Kompleksitas Memori

- Untuk menyimpan *dist[u][v][l]*
  - $\rightarrow$ **O(|V| x |V| x |V|) = O(|V|^3)**

Algoritma 2: Algoritma Shimbel

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \rightarrow v)$
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
```

Can we do better?

Faktanya, kita bisa mengeliminasi indeks l sehingga memori yang digunakan adalah **O(|V|^2)**.

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* $u, v \in V$

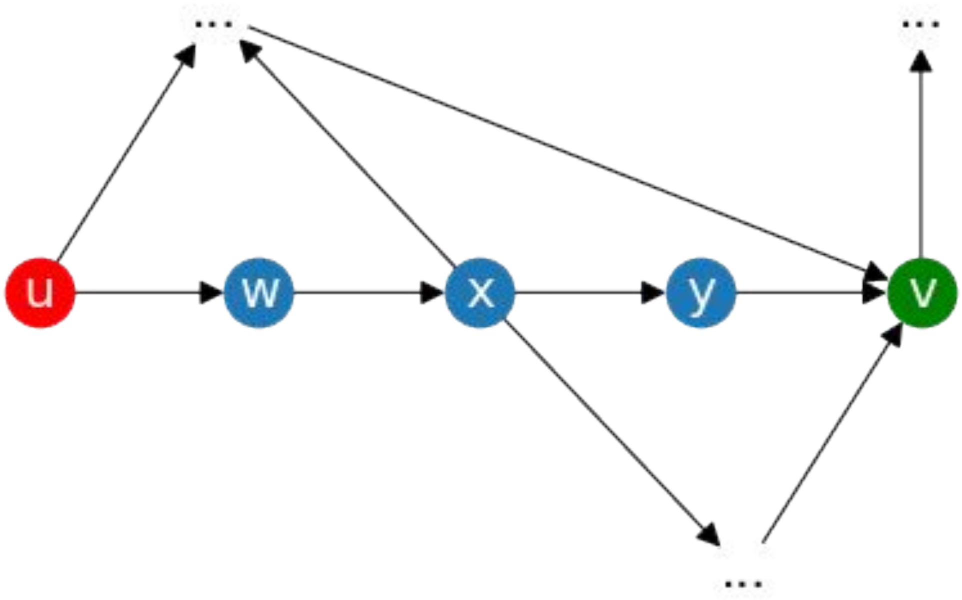
foreach $u \in V$ **do**
  foreach $v \in V$ **do**
    if $u = v$ **then** $dist[u][v] \leftarrow 0$
    else $dist[u][v] \leftarrow \infty$
  end
end

foreach $l \leftarrow 1$ to $|V| - 1$ **do**
  foreach $u \in V$ **do**
    foreach $(x \to v) \in E$ **do**
      $dist[u][v] \leftarrow \min(dist[u][v], dist[u][x] + w(x \to v))$
    end
  end
end

return $dist$

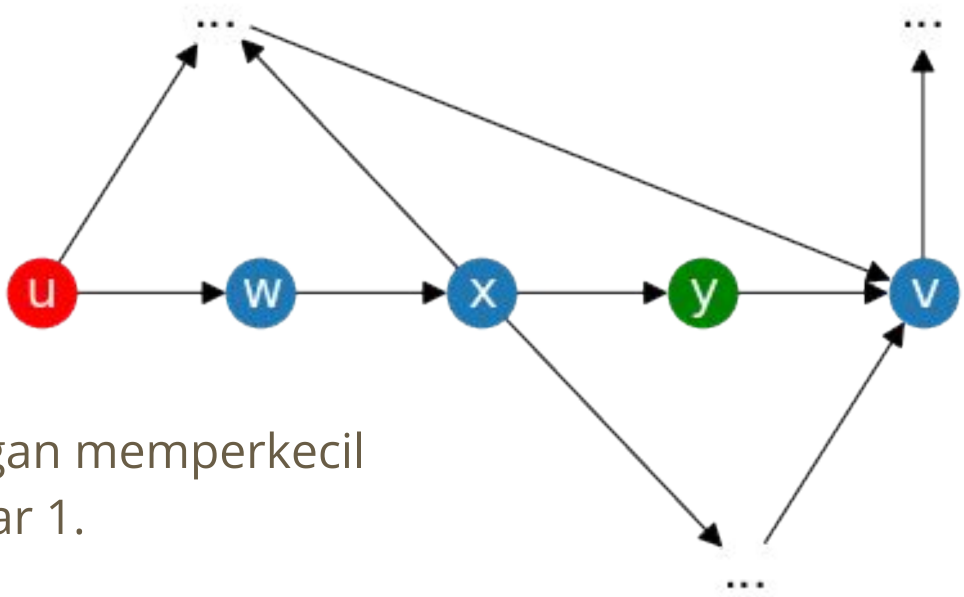
Dynamic Programming + Divide and Conquer

Saat kita menghitung *dist(u,v,l)* pada suatu graf...



Dynamic Programming + Divide and Conquer

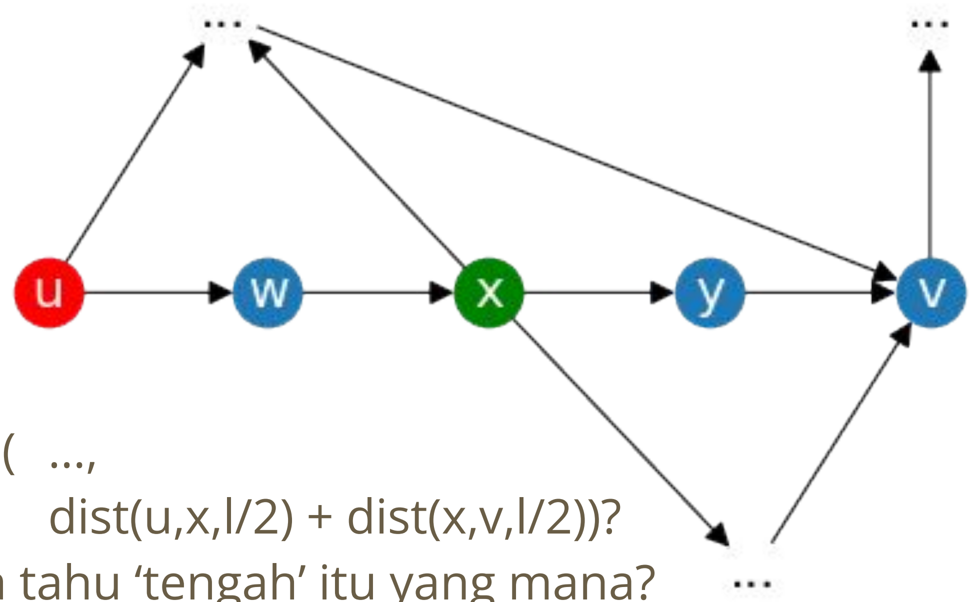
Kita akan melakukan rekursi ke *node* yang bisa mencapai v dalam 1 langkah.



Sama saja dengan memperkecil masalah sebesar 1.

Dynamic Programming + Divide and Conquer

Bagaimana kalau langsung '*loncat ke tengah*'?



dist(u,v,l) = min(...,
dist(u,x,l/2) + dist(x,v,l/2))?

Bagaimana kita tahu 'tengah' itu yang mana?

Dynamic Programming + Divide and Conquer

Bagaimana caranya kita tahu ‘tengah’ itu yang mana?

Jawabannya: Coba saja semua *node* sebagai ‘tengah’-nya!

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

dist(u,v) yang kita inginkan tetap berada pada **dist(u, v, |V| - 1)**.

Dynamic Programming + Divide and Conquer

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

Satu masalah: Misalkan |V| = 6. Jawaban kita berada pada dist(u, v, 6 - 1).

dist(u, v, 5) = min_x(dist(u, x, **2.5**) + dist(x, v, **2.5**)) ?

dist(u, v, l) tidak terdefinisi apabila l bukan bilangan bulat!
Faktanya, dist(u, v, l) hanya terdefinisi apabila l merupakan bilangan dua pangkat.

Dynamic Programming + Divide and Conquer

Ingat kembali observasi bahwa *shortest path* pada graf tanpa *negative cycle* pasti menggunakan *edge* paling banyak |V| - 1 *edge*.

Artinya, **dist(u, v, |V| - 1) = dist(u, v, |V|) = dist(u, v, |V| + 1) = ...**

Maka, kita bisa mengatakan bahwa dist(u, v) = dist(u, v, L)
dimana L merupakan bilangan dua pangkat lebih besar (atau sama dengan) |V| - 1.

Kita bisa mengambil nilai $L = 2^{\text{ceil}(\lg |V|)}$ sebagai bilangan yang terdekat dengan |V|.

Dynamic Programming + Divide and Conquer

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

dist(u,v) yang kita inginkan tetap berada pada **dist(u, v, L)** dengan **L = $2^{\text{ceil(lg }|V|)}$**.

Rekurens tersebut bisa dihitung dengan *dynamic programming*.

Dynamic Programming + Divide and Conquer

Rekurens tersebut bisa dihitung dengan *dynamic programming*.

Dengan catatan bahwa memo yang kita isi adalah dist[u][v][i] yaitu dist(u, v, 2^i) karena yang terdefinisi hanyalah dist(u, v, l) dengan l bilangan dua pangkat.

- Mulai dari *base case* i = 0, lalu
- Isi tabel *memo* dari i = 1, 2, 3, sampai ceil(lg |V|)

Algoritma ini dinamakan **algoritma Fischer-Meyer.**

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \le i \le \lceil \lg |V| \rceil)$

```
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
```

Fischer-Meyer -- Kompleksitas Waktu

- *For loop* pertama untuk mengisi *base case*
  - → **O(|V|²)**
- *For loop* kedua untuk mengisi tabel keseluruhan
  - → **O(lg |V| x |V| x |V| x |V|)**
  - **= O(|V|³ log |V|)**

Kompleksitas waktu total = **O(|V|³ log |V|)**

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \le i \le \lceil \lg |V| \rceil)$

```
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
```

Fischer-Meyer -- Kompleksitas Memori

- Untuk menyimpan *dist[u][v][l]*
  - → **O(|V| x |V| x |V|) = O(|V|^3)**

Can we do better (again)?

Kita juga bisa mengeliminasi indeks l sehingga memori yang digunakan adalah **O(|V|^2)**.
(Leyzorek et al., 1957)

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \le i \le \lceil \lg |V| \rceil)$

foreach $u \in V$ **do**
  foreach $v \in V$ **do**
    $dist[u][v][0] \leftarrow w(u \to v)$
  end
end

foreach $i \leftarrow 1$ to $\lceil \lg |V| \rceil$ **do**
  foreach $u \in V$ **do**
    foreach $v \in V$ **do**
      $dist[u][v][i] \leftarrow \infty$
      foreach $x \in V$ **do**
        $dist[u][v][i] \leftarrow$
          $\min(dist[u][v][i],\ dist[u][x][i-1] + dist[x][v][i-1])$
      end
    end
  end
end

return *dist*

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
```

Fischer-Meyer -- Perkalian Matriks

Bandingkan *inner loop* pada Fischer-Meyer dengan perkalian matriks persegi.

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$

keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \leq i \leq \lceil \lg |V| \rceil)$

```
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
```

Fischer-Meyer -- Perkalian Matriks

Bandingkan *inner loop* pada Fischer-Meyer dengan perkalian 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
```

```
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
```

Fischer-Meyer -- Perkalian Matriks

Bandingkan *inner loop* pada Fischer-Meyer dengan perkalian 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 $i \leftarrow 1$ *to* n **do**
  foreach $j \leftarrow 1$ *to* n **do**
    $AB[i][j] \leftarrow 0$
    foreach $k \leftarrow 1$ *to* n **do**
      $AB[i][j] \leftarrow AB[i][j] + A[i][k] \times B[k][j]$
    end
  end
end

return AB

---

foreach $\boxed{u \in V}$ **do** loop i
  foreach $\boxed{v \in V}$ **do** loop j
    $dist[u][v][i] \leftarrow \infty$ inisialisasi AB[i][j]
    foreach $\boxed{x \in V}$ **do** loop k
      $\boxed{dist[u][v][i] \leftarrow \min(dist[u][v][i],\ dist[u][x][i-1] + dist[x][v][i-1])}$
    end
  end
end

(+) (×)

AB[i][j] ← AB[i][j] + A[i][k] × B[k][j]

Fischer-Meyer -- Perkalian Matriks

Kita bisa menganggap pada algoritma Fischer-Meyer, dist[•][•][i] dihasilkan dengan ‘mengalikan’ dist[•][•][i-1] dengan dirinya sendiri.

Operasi ‘perkalian matriks’ ini cukup spesial, karena operasi (+) diganti dengan operasi min, dan operasi (×) diganti dengan operasi (+).

foreach $\boxed{u \in V}$ **do** loop i
  foreach $\boxed{v \in V}$ **do** loop j
    $dist[u][v][i] \leftarrow \infty$ inisialisasi AB[i][j]
    foreach $\boxed{x \in V}$ **do** loop k
      $\boxed{dist[u][v][i] \leftarrow \min(dist[u][v][i],\ dist[u][x][i-1] + dist[x][v][i-1])}$
    end
  end
end

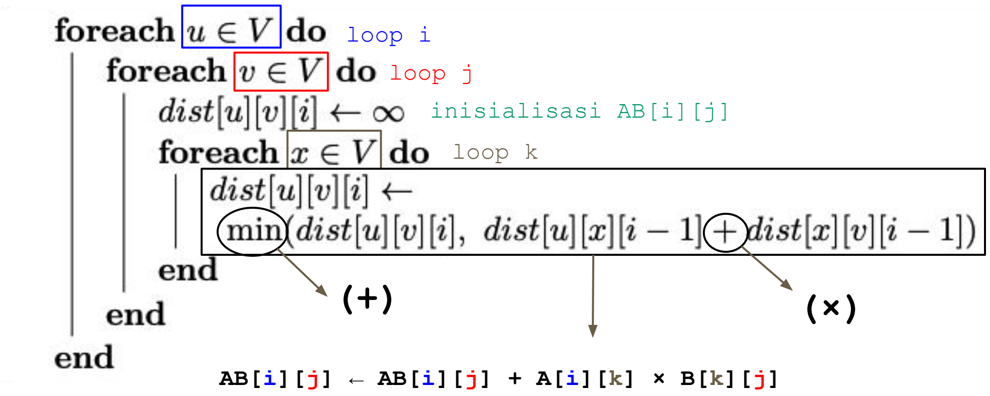
(+)

(×)

AB[i][j] ← AB[i][j] + A[i][k] × B[k][j]

Fischer-Meyer -- Perkalian Matriks

Operasi ‘perkalian matriks’ ini cukup spesial, karena operasi (+) diganti dengan operasi min, dan operasi (×) diganti dengan operasi (+).



Karena spesial, kita beri nama operasi ⊗ atau perkalian matriks *min-plus*.

Fischer-Meyer -- Perkalian Matriks

Kita bisa mengganti *inner loop* tersebut dengan:

dist[•][•][i]

← dist[•][•][i-1] ⊗ dist[•][•][i-1]

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$

keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \leq i \leq \lceil \lg |V| \rceil)$

foreach $u \in V$ **do**
    foreach $v \in V$ **do**
        $dist[u][v][0] \leftarrow w(u \to v)$
    end
end

foreach $i \leftarrow 1$ to $\lceil \lg |V| \rceil$ **do**
    foreach $u \in V$ **do**
        foreach $v \in V$ **do**
            $dist[u][v][i] \leftarrow \infty$
            foreach $x \in V$ **do**
                $dist[u][v][i] \leftarrow$
                  $\min(dist[u][v][i],\ dist[u][x][i-1] + dist[x][v][i-1])$
            end
        end
    end
end

return *dist*

Fischer-Meyer -- Perkalian Matriks

Kita bisa mengganti *inner loop* tersebut dengan:

dist[•][•][i]
    ← dist[•][•][i-1] ⊗ dist[•][•][i-1]

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \leq i \leq \lceil \lg |V| \rceil)$

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

foreach i ← 1 to ⌈lg |V|⌉ do
    dist[•][•][i]
     ← dist[•][•][i-1] ⊗ dist[•][•][i-1]
end

return dist
```

Algoritma Shimbel -- Perkalian Matriks

Tidak hanya algoritma Fischer-Meyer, algoritma Shimbel pun sebenarnya menggunakan operasi yang sama!

Namun, tidak langsung jelas terlihat..

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
```

Algoritma Shimbel -- Perkalian Matriks

Tidak hanya algoritma Fischer-Meyer, algoritma Shimbel pun sebenarnya menggunakan operasi yang sama!

Namun, tidak langsung jelas terlihat..

Inner loop di samping sebenarnya melakukan 'perkalian' yang sama, namun dengan beberapa iterasi disingkat karena nilainya sudah jelas..

foreach $u \in V$ **do**
  foreach $v \in V, v \neq u$ **do**
    $dist[u][v][l] \leftarrow dist[u][v][l-1]$
    foreach $(x \rightarrow v) \in E$ **do**
      $dist[u][v][l] \leftarrow \min(dist[u][v][l], dist[u][x][l-1] + w(x \rightarrow v))$
    end
  end
end

Algoritma Shimbel -- Perkalian Matriks

Inner loop di samping sebenarnya melakukan ‘perkalian’ yang sama, namun dengan beberapa iterasi disingkat karena nilainya sudah jelas..

Misalkan **W[u][v] = w(u → v)**. W adalah *adjacency matrix* pada graf tersebut.

Ternyata, dist[•][•][l] didapat dengan ‘mengalikan’ dist[•][•][l-1] dengan W[•][•]!

loop j, namun u = v dilewatkan karena dist[u][v][l] sudah pasti nol.

inisialisasi dist[u][v][l], namun langsung dengan dist[u][v][l-1] karena sama saja dengan menggunakan x = v,
dist[u][v][l-1] = dist[u][v][l-1] + w(v → v)

foreach $\boxed{u \in V}$ **do** loop i
  foreach $\boxed{v \in V, v \neq u}$ **do**
    $dist[u][v][l] \leftarrow dist[u][v][l-1]$
    foreach $\boxed{(x \to v) \in E}$ **do** loop k, melewati *node* yg tidak bisa mencapai v
      $\boxed{dist[u][v][l] \leftarrow \min(dist[u][v][l], dist[u][x][l-1] + w(x \to v))}$
    end
  end
end

W[u][v]

```
dist[u][v][l] ←
  min(dist[u][v][l], dist[u][x][l-1] + W[x][v])
```

Algoritma Shimbel -- Perkalian Matriks

Kita bisa mengganti *inner loop* tersebut dengan:

dist[•][•][l]
← dist[•][•][l-1] ⊗ W[•][•]

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 $u \in V$ **do**
  foreach $v \in V$ **do**
    if $u = v$ **then** $dist[u][v][0] \leftarrow 0$
    else $dist[u][v][0] \leftarrow \infty$
  end
end

foreach $l \leftarrow 1$ to $|V| - 1$ **do**
  foreach $u \in V$ **do**
    foreach $v \in V, v \neq u$ **do**
      $dist[u][v][l] \leftarrow dist[u][v][l-1]$
      foreach $(x \to v) \in E$ **do**
        $dist[u][v][l] \leftarrow \min(dist[u][v][l], dist[u][x][l-1] + w(x \to v))$
      end
    end
  end
end

return *dist*

Algoritma Shimbel -- Perkalian Matriks

Kita bisa mengganti *inner loop* tersebut dengan:

dist[•][•][l]
    ← dist[•][•][l-1] ⊗ W[•][•]

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 $u \in V$ **do**
  foreach $v \in V$ **do**
    if $u = v$ **then** $dist[u][v][0] \leftarrow 0$
    else $dist[u][v][0] \leftarrow \infty$
  end
end

foreach $l \leftarrow 1$ to $|V| - 1$ **do**

```
dist[•][•][l]
 ← dist[•][•][l-1] ⊗ W[•][•]
```

end

return *dist*

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
    dist[•][•][l]
     ← dist[•][•][l-1] ⊗ W[•][•]
end

return dist
```

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \leq i \leq \lceil \lg |V| \rceil)$

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

foreach i ← 1 to ⌈lg |V|⌉ do
    dist[•][•][i]
     ← dist[•][•][i-1] ⊗ dist[•][•][i-1]
end

return dist
```

Kedua algoritma ini mungkin memberikan hasil (dist) yang mempunyai arti yang sedikit berbeda...

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
```

Matriks identitas dalam *min-plus*:

$I = W^0$

```
foreach l ← 1 to |V| − 1 do
    dist[•][•][l]
     ← dist[•][•][l-1] ⊗ W[•][•]
end

return dist
```

dist(u,v) = dist[u][v][|V|-1] = $W^{|V|-1}$

Namun, sebenarnya yang dihasilkan adalah sama!

Algoritma 4: Algoritma Fischer-Meyer untuk APSP

masukan: graf *weighted* $G = (V, E)$ dan fungsi *weight* $w(u \to v)$
keluaran: $dist[u][v][i]$ = panjang *shortest path* $u \Rightarrow v$ untuk setiap pasang *node* $u, v \in V$ dengan menggunakan paling banyak 2^i *edge* $(0 \le i \le \lceil \lg |V| \rceil)$

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

dist[u][v][0] = $W^{2\wedge 0} = W^1 = \mathbf{W}$

```
foreach i ← 1 to ⌈lg |V|⌉ do
    dist[•][•][i]
     ← dist[•][•][i-1] ⊗ dist[•][•][i-1]
end

return dist
```

dist(u,v) = dist[u][v][ceil(lg|V|)] = $W^{2\wedge ceil(\lg |V|)}$ = $W^{|V|*}$

***approx**

Perkalian Matriks -- Observasi

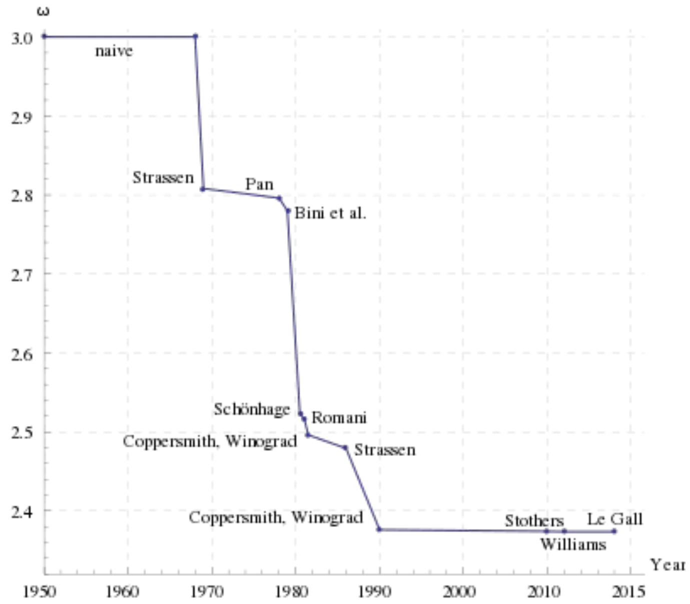
- **Algoritma Shimbel** sebenarnya menghasilkan $W^{|V|-1}$.
- **Algoritma Fischer** juga demikian, namun mempercepat perpangkatan menggunakan ***exponentiation by squaring***.
  - ‘Perkalian matriks’ ⊗ bersifat asosiatif, sama seperti perkalian matriks biasa.

- Kedua algoritma mengalikan matriks dengan cara *naive* (metode baris-kolom).
  - Pertanyaannya: **Apakah bisa dipercepat?**

Perkalian Matriks -- Observasi

- Kedua algoritma mengalikan matriks dengan cara *naive* (metode baris-kolom).
  - Pertanyaannya: **Apakah bisa dipercepat?**

- Sebelumnya, kita tahu beberapa cara mengalikan matriks (biasa) dengan cepat seperti algoritma Strassen dengan kompleksitas $O(n^{\lg 7})$
  - Apakah bisa diaplikasikan ke perkalian ⊗ ?



* Gambar diambil dari Wikipedia, https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

Perkalian Matriks -- Strassen?

Apakah algoritma Strassen bisa diaplikasikan ke perkalian ⊗ ?

```
STRASSEN(A, B)
// A, B = matriks persegi yang akan dihitung hasil perkaliannya (AB)

1.  N = length(A)
2.  if N == 1
3.      return A x B // Perkalian naif

// submatrix(A, r1, c1, r2, c2) adalah fungsi
// yang mengembalikan submatriks dari A
// dengan ukuran (r2-r1) x (c2-c1) dan
// isi A_{i,j} dengan r1 ≤ i < r2 && c1 ≤ j < c2

4.  A_{0,0} <- submatrix(A, 0, 0, N / 2, N / 2)
5.  A_{0,1} <- submatrix(A, 0, N / 2, N / 2, N)
6.  A_{1,0} <- submatrix(A, N / 2, 0, N, N / 2)
7.  A_{1,1} <- submatrix(A, N / 2, N / 2, N, N)

8.  B_{0,0} <- submatrix(B, 0, 0, N / 2, N / 2)
9.  B_{0,1} <- submatrix(B, 0, N / 2, N / 2, N)
10. B_{1,0} <- submatrix(B, N / 2, 0, N, N / 2)
11. B_{1,1} <- submatrix(B, N / 2, N / 2, N, N)

12. M_0 <- STRASSEN(A_{0,0} + A_{1,1}, B_{0,0} + B_{1,1})
13. M_1 <- STRASSEN(A_{1,0} + A_{1,1}, B_{0,0})
14. M_2 <- STRASSEN(A_{0,0}, B_{0,1} - B_{1,1})
15. M_3 <- STRASSEN(A_{1,1}, B_{1,0} - B_{0,0})
16. M_4 <- STRASSEN(A_{0,0} + A_{0,1}, B_{1,1})
17. M_5 <- STRASSEN(A_{1,0} - A_{0,0}, B_{0,0} + B_{0,1})
18. M_6 <- STRASSEN(A_{0,1} - A_{1,1}, B_{1,0} + B_{1,1})

19. C_{0,0} <- M_0 + M_3 - M_4 + M_6
20. C_{0,1} <- M_2 + M_4
21. C_{1,0} <- M_1 + M_3
22. C_{1,1} <- M_0 - M_1 + M_2 + M_5

23. C <- Matriks nol ukuran N x N
24. for i <- 0 to N/2 - 1 do
25.   for j <- 0 to N/2 - 1 do
26.     C[i][j] <- C_{1,1}[i][j]
27. for i <- 0 to N/2 - 1 do
28.   for j <- 0 to N/2 - 1 do
29.     C[i][j + N/2] <- C_{1,2}[i][j]
30. for i <- 0 to N/2 - 1 do
31.   for j <- 0 to N/2 - 1 do
32.     C[i + N/2][j] <- C_{2,1}[i][j]
33. for i <- 0 to N/2 - 1 do
34.   for j <- 0 to N/2 - 1 do
35.     C[i + N/2][j + N/2] <- C_{2,2}[i][j]

36. return C
```

**pseudocode* diambil dari laporan TK1 kelompok sad B01

Perkalian Matriks -- Strassen?

Apakah algoritma Strassen bisa diaplikasikan ke perkalian ⊗ ?

Karena perkalian ⊗ bisa dikatakan sebagai perkalian matriks di ‘alam’ dimana penjumlahan skalar diganti dengan operasi min, dan perkalian skalar diganti dengan operasi (+), maka kita harus menyesuaikan algoritma Strassen dengan alam tersebut.

```
STRASSEN(A, B)
// A, B = matriks persegi yang akan dihitung hasil perkaliannya (AB)

1.  N = length(A)
2.  if N == 1
3.      return A x B // Perkalian naif

// submatrix(A, r1, c1, r2, c2) adalah fungsi
// yang mengembalikan submatriks dari A
// dengan ukuran (r2-r1) x (c2-c1) dan
// isi A_{i,j} dengan r1 ≤ i < r2 && c1 ≤ j < c2

4.  A_{0,0} <- submatrix(A, 0, 0, N / 2, N / 2)
5.  A_{0,1} <- submatrix(A, 0, N / 2, N / 2, N)
6.  A_{1,0} <- submatrix(A, N / 2, 0, N, N / 2)
7.  A_{1,1} <- submatrix(A, N / 2, N / 2, N, N)

8.  B_{0,0} <- submatrix(B, 0, 0, N / 2, N / 2)
9.  B_{0,1} <- submatrix(B, 0, N / 2, N / 2, N)
10. B_{1,0} <- submatrix(B, N / 2, 0, N, N / 2)
11. B_{1,1} <- submatrix(B, N / 2, N / 2, N, N)

12. M_0 <- STRASSEN(A_{0,0} + A_{1,1}, B_{0,0} + B_{1,1})
13. M_1 <- STRASSEN(A_{1,0} + A_{1,1}, B_{0,0})
14. M_2 <- STRASSEN(A_{0,0}, B_{0,1} - B_{1,1})
15. M_3 <- STRASSEN(A_{1,1}, B_{1,0} - B_{0,0})
16. M_4 <- STRASSEN(A_{0,0} + A_{0,1}, B_{1,1})
17. M_5 <- STRASSEN(A_{1,0} - A_{0,0}, B_{0,0} + B_{0,1})
18. M_6 <- STRASSEN(A_{0,1} - A_{1,1}, B_{1,0} + B_{1,1})

19. C_{0,0} <- M_0 + M_3 - M_4 + M_6
20. C_{0,1} <- M_2 + M_4
21. C_{1,0} <- M_1 + M_3
22. C_{1,1} <- M_0 - M_1 + M_2 + M_5

23. C <- Matriks nol ukuran N x N
24. for i <- 0 to N/2 - 1 do
25.   for j <- 0 to N/2 - 1 do
26.     C[i][j] <- C_{1,1}[i][j]
27. for i <- 0 to N/2 - 1 do
28.   for j <- 0 to N/2 - 1 do
29.     C[i][j + N/2] <- C_{1,2}[i][j]
30. for i <- 0 to N/2 - 1 do
31.   for j <- 0 to N/2 - 1 do
32.     C[i + N/2][j] <- C_{2,1}[i][j]
33. for i <- 0 to N/2 - 1 do
34.   for j <- 0 to N/2 - 1 do
35.     C[i + N/2][j + N/2] <- C_{2,2}[i][j]

36. return C
```

**pseudocode* diambil dari laporan TK1 kelompok sad B01

Perkalian Matriks -- Strassen?

Karena perkalian $\otimes$ bisa dikatakan sebagai perkalian matriks di ‘alam’ dimana penjumlahan skalar diganti dengan operasi min, dan perkalian skalar diganti dengan operasi (+), maka kita harus menyesuaikan algoritma Strassen dengan alam tersebut.

```
M_0 <- STRASSEN(A_{0,0} + A_{1,1}, B_{0,0} + B_{1,1})
M_1 <- STRASSEN(A_{1,0} + A_{1,1}, B_{0,0})
M_2 <- STRASSEN(A_{0,0}, B_{0,1} - B_{1,1})
M_3 <- STRASSEN(A_{1,1}, B_{1,0} - B_{0,0})
M_4 <- STRASSEN(A_{0,0} + A_{0,1}, B_{1,1})
M_5 <- STRASSEN(A_{1,0} - A_{0,0}, B_{0,0} + B_{0,1})
M_6 <- STRASSEN(A_{0,1} - A_{1,1}, B_{1,0} + B_{1,1})

C_{0,0} <- M_0 + M_3 - M_4 + M_6
C_{0,1} <- M_2 + M_4
C_{1,0} <- M_1 + M_3
C_{1,1} <- M_0 - M_1 + M_2 + M_5
```

**pseudocode* diambil dari laporan TK1 kelompok sad B01

Perkalian Matriks -- Strassen?

Sayangnya, tidak ada operasi yang ekuivalen dengan pengurangan pada dunia tersebut. (operasi min tidak punya invers).

Maka, algoritma Strassen tidak bisa diaplikasikan pada operasi $\otimes$.

$M_0 \leftarrow \text{STRASSEN}(A_{0,0} + A_{1,1},\ B_{0,0} + B_{1,1})$
$M_1 \leftarrow \text{STRASSEN}(A_{1,0} + A_{1,1},\ B_{0,0})$
$M_2 \leftarrow \text{STRASSEN}(A_{0,0},\ B_{0,1} - B_{1,1})$
$M_3 \leftarrow \text{STRASSEN}(A_{1,1},\ B_{1,0} - B_{0,0})$
$M_4 \leftarrow \text{STRASSEN}(A_{0,0} + A_{0,1},\ B_{1,1})$
$M_5 \leftarrow \text{STRASSEN}(A_{1,0} - A_{0,0},\ B_{0,0} + B_{0,1})$
$M_6 \leftarrow \text{STRASSEN}(A_{0,1} - A_{1,1},\ B_{1,0} + B_{1,1})$

$C_{0,0} \leftarrow M_0 + M_3 - M_4 + M_6$
$C_{0,1} \leftarrow M_2 + M_4$
$C_{1,0} \leftarrow M_1 + M_3$
$C_{1,1} \leftarrow M_0 - M_1 + M_2 + M_5$

**pseudocode* diambil dari laporan TK1 kelompok sad B01

Perkalian Matriks dan Graf

- Sebenarnya, beberapa permasalahan graf bisa diselesaikan dengan matriks.
  - Seperti contoh yang telah ditunjukkan sebelumnya
  - Masih banyak lagi, seperti menghitung banyak kemungkinan *path* u ⇒ v pada graf, atau mendeteksi apakah ada *cycle* dengan panjang k atau tidak.

- Ada algoritma untuk mencari APSP pada graf *undirected* dengan memanfaatkan perkalian matriks biasa (bukan *min-plus*).
  - Ditemukan oleh Raimund Seidel dengan *expected time complexity* $O(M(|V|) \log |V|)$ dengan $M(n) = O(n^{2.37})$ (untuk mengalikan matriks)

Sekarang, kembali ke pendekatan rekursif...

*src:
https://me.me/i/a-letter-for-you-u-have-scrolled-to-far-its-bc1e90c0d25a4700bbeed812249b1265

Pendekatan Rekursif -- Formulasi Lain

Ingat kembali definisi rekursif dist(u,v,l):
dist(u,v,l) = Panjang *shortest path* u ⇒ v dengan menggunakan paling banyak *l* *edge*.

Sekarang, kita akan menggunakan formulasi lain:
dist(u,v,r) = Panjang *shortest path* u ⇒ v **dengan 'melewati' *node* dengan nomor kurang (atau sama dengan) dari r**.

Untuk ini, mari asumsikan *node* pada graf dinomori 1 sampai |V| (bebas).

Pendekatan Rekursif -- Formulasi Lain

dist(u,v,r) = Panjang *shortest path* u ⇒ v **dengan 'melewati'** ***node*** **dengan nomor kurang (atau sama dengan) dari r**.

Apa yang dimaksud dengan '**melewati**'?

Contoh, pada path 0 ⇒ 5:

0 → **1** → **2** → **3** → **4** → 5

Path tersebut dikatakan diawali dengan 0, **melewati 1, 2, 3, dan 4** dan diakhiri dengan 5.

Pendekatan Rekursif -- Formulasi Lain

dist(u,v,r) = Panjang *shortest path* u ⇒ v **dengan 'melewati'** ***node*** **dengan nomor kurang (atau sama dengan) dari r**.

Bagaimana dengan rekurensnya?

Pendekatan Rekursif -- Formulasi Lain

Bagaimana dengan rekurensnya?

Misalkan kita ingin mencari dist(u,v,r).
Ada dua kemungkinan:

- *Shortest path* u ⇒ v **melewati *node* bernomor r**.
  - *path* antara u ⇒ r dan r ⇒ v pasti melewati *node* bernomor kurang dari r.
- *Shortest path* u ⇒ **tidak melewati *node* bernomor r.**
  - *path* u ⇒ v pasti melewati *node* bernomor kurang dari r.

Pendekatan Rekursif -- Formulasi Lain

Ada dua kemungkinan:

- *Shortest path* u ⇒ v **melewati *node* bernomor r**.
  - *path* antara u ⇒ r dan r ⇒ v pasti melewati *node* bernomor kurang dari r.
- *Shortest path* u ⇒ **tidak melewati *node* bernomor r.**
  - *path* u ⇒ v pasti melewati *node* bernomor kurang dari r.

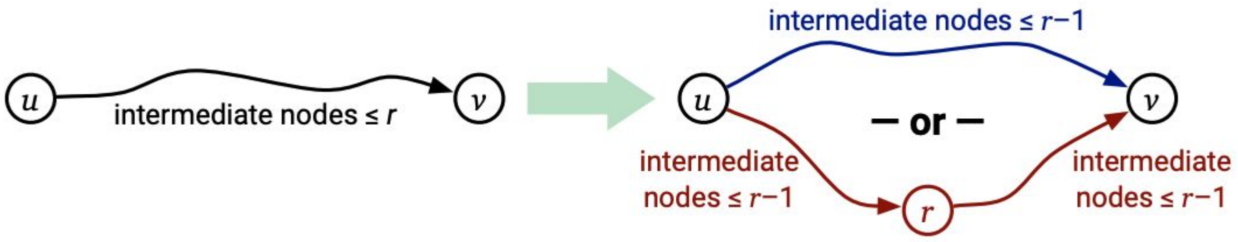


Figure 9.3. Recursive structure of the restricted shortest path $\pi(u, v, r)$.

*gambar diambil dari buku *Algorithms* (Jeff Erickson)

Pendekatan Rekursif -- Formulasi Lain

Ada dua kemungkinan:

- *Shortest path* u ⇒ v **melewati *node* bernomor r**.
    - *path* antara u ⇒ r dan r ⇒ v pasti melewati *node* bernomor kurang dari r.
- *Shortest path* u ⇒ **tidak melewati *node* bernomor r.**
    - *path* u ⇒ v pasti melewati *node* bernomor kurang dari r.

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

dist(u, v) yang kita cari (tentu saja) terdapat pada dist(u, v, |V|).

Pendekatan Rekursif -- Algoritma Kleene

Rekurens tersebut bisa dihitung dengan *dynamic programming*.

- Mulai dari *base case* r = 0, lalu
- Isi tabel *memo* dari r = 1, 2, 3, sampai |V|

Algoritma ini disebut dengan **algoritma Kleene**.

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
```

Kleene -- Kompleksitas Waktu

- *For loop* pertama untuk mengisi *base case*
  - → **O(|V|²)**
- *For loop* kedua untuk mengisi tabel keseluruhan
  - → **O(|V| x |V| x |V|) = O(|V|³)**

Kompleksitas waktu total = **O(|V|³)**

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
```

Kleene -- Kompleksitas Memori

- Untuk menyimpan *dist[u][v][l]*
  - → **O(|V| x |V| x |V|) = O(|V|^3)**

Can we do better (again...)?

Kita juga bisa mengeliminasi indeks r sehingga memori yang digunakan adalah **O(|V|^2)**. (Floyd-Warshall)

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
```

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

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

return dist
```

All-Pairs Shortest Path -- Mencari *pred*

Ingat kembali definisi permasalahan APSP.

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.

Algoritma-algoritma sebelumnya **belum menemukan nilai *pred*!**

All-Pairs Shortest Path -- Mencari *pred*

Algoritma-algoritma sebelumnya **belum menemukan nilai *pred*!**

Untungnya, apabila kita mengetahui nilai *dist*, pasti kita bisa menemukan nilai *pred*.

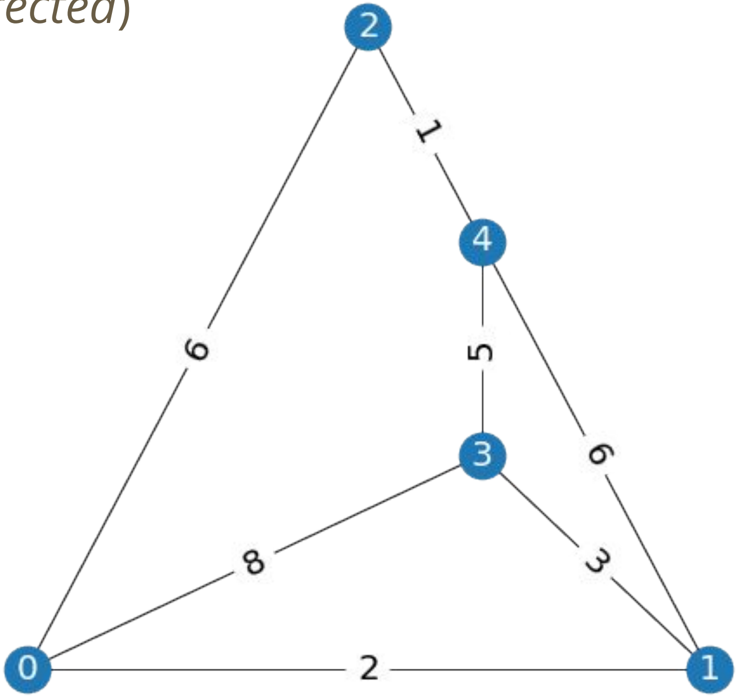
(salah satu) idenya adalah sebagai berikut:

- Untuk setiap *edge* x → v, cari *shortest path* u ⇒ v mana saja yang menggunakan *edge* x → v untuk mencapai v. Apabila terdapat *shortest path* tersebut, maka *pred(u,v) = x*.
  - *edge* x → v dikatakan sebagai 'digunakan' apabila **dist(u,v) = dist(u,x) + w(x → v)**

All-Pairs Shortest Path -- Mencari *pred*

Contoh G = (V, E) (*undirected*)

		v				
u		0	1	2	3	4
	0	**0**	**2**	**6**	**5**	**7**
	1	**2**	**0**	**7**	**3**	**6**
	2	**6**	**7**	**0**	**6**	**1**
	3	**5**	**3**	**6**	**0**	**5**
	4	**7**	**6**	**1**	**5**	**0**
dist(u,v)						

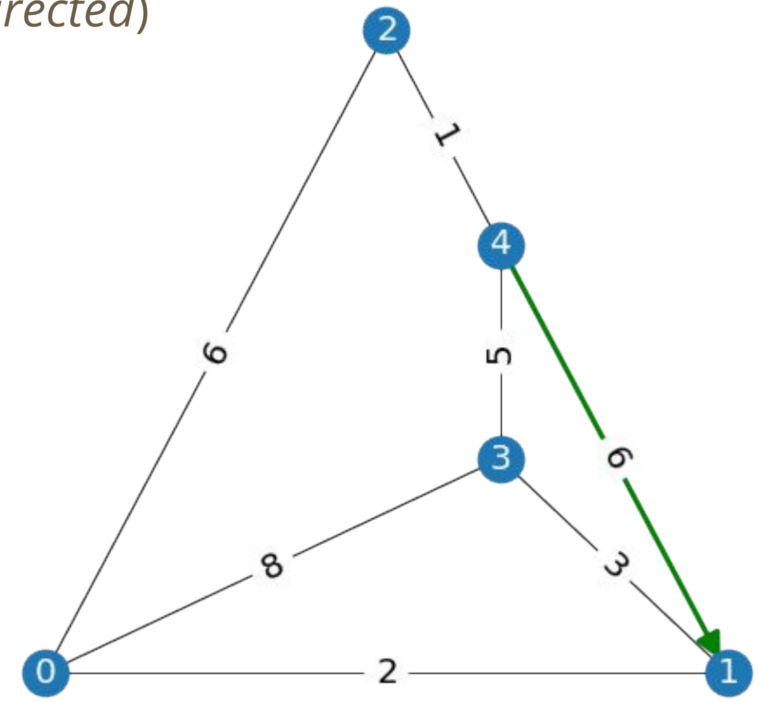


All-Pairs Shortest Path -- Mencari *pred*

Contoh G = (V, E) (*undirected*)

u \ v	0	1	2	3	4
0	**0**	**2**	**6**	**5**	**7**
1	**2**	**0**	**7**	**3**	**6**
2	**6**	**7**	**0**	**6**	**1**
3	**5**	**3**	**6**	**0**	**5**
4	**7**	**6**	**1**	**5**	**0**

dist(u,v)



Ambil *edge* 4 → 1.

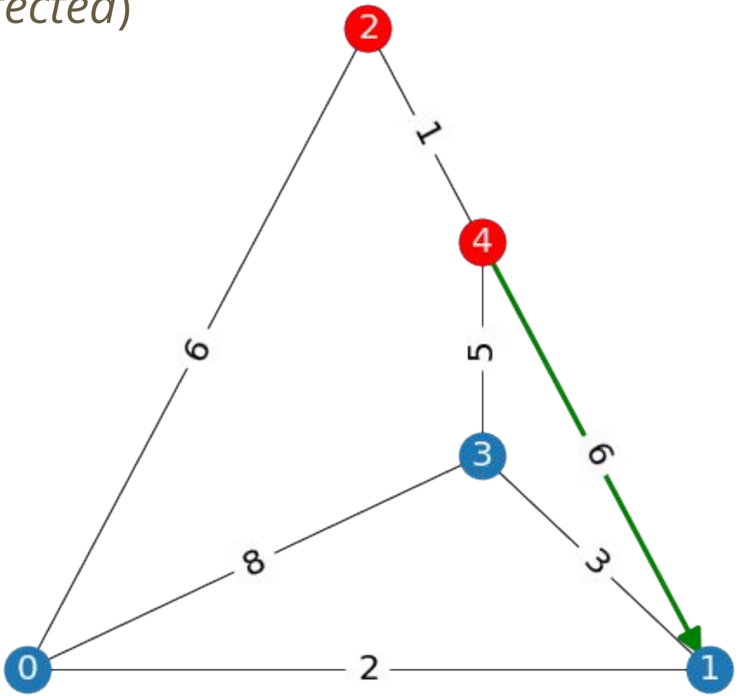
Kita ingin mencari *shortest path* u ⇒ 1 mana saja yang menggunakan *edge* 4 → 1.

All-Pairs Shortest Path -- Mencari *pred*

Contoh G = (V, E) (*undirected*)

	v					
u		0	1	2	3	4
	0	**0**	**2**	**6**	**5**	**7**
	1	**2**	**0**	**7**	**3**	**6**
	2	**6**	**7**	**0**	**6**	**1**
	3	**5**	**3**	**6**	**0**	**5**
	4	**7**	**6**	**1**	**5**	**0**

dist(u,v)



Ambil *edge* 4 → 1.

Kita ingin mencari *shortest path* u ⇒ 1 mana saja yang menggunakan *edge* 4 → 1.

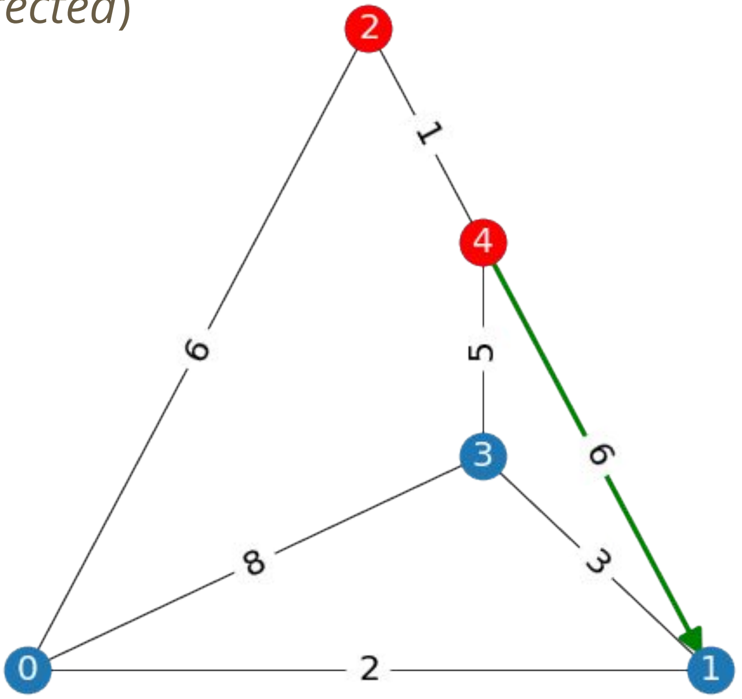
Shortest path
2 ⇒ 1 dan 4 ⇒ 1
menggunakan *edge* 4 → 1!

dist(2,1) = dist(2,4) + w(4 → 1)
dist(4,1) = dist(4,4) + w(4 → 1)

All-Pairs Shortest Path -- Mencari *pred*

Contoh G = (V, E) (*undirected*)

	v					
u		0	1	2	3	4
	0	**0**	**2**	**6**	**5**	**7**
	1	**2**	**0**	**7**	**3**	**6**
	2	**6**	**7**	**0**	**6**	**1**
	3	**5**	**3**	**6**	**0**	**5**
	4	**7**	**6**	**1**	**5**	**0**
dist(u,v)						



Ambil *edge* 4 → 1.

Kita ingin mencari *shortest path* u ⇒ 1 mana saja yang menggunakan *edge* 4 → 1.

Shortest path
2 ⇒ 1 dan 4 ⇒ 1
menggunakan *edge* 4 → 1!

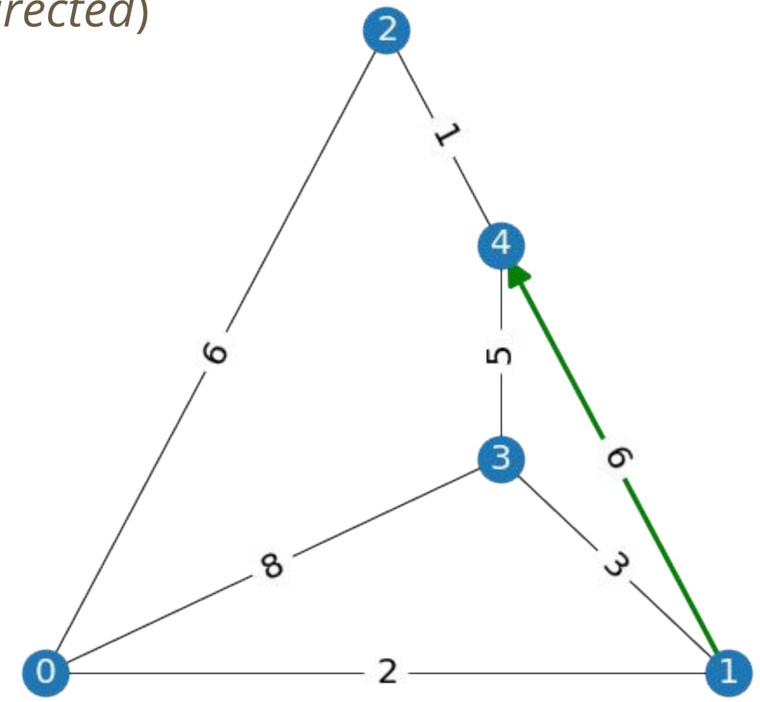
dist(2,1) = dist(2,4) + w(4 → 1)
dist(4,1) = dist(4,4) + w(4 → 1)

Artinya,
pred(2,1) ← 4
pred(4,1) ← 4

All-Pairs Shortest Path -- Mencari *pred*

Contoh G = (V, E) (*undirected*)

		v				
u		0	1	2	3	4
	0	**0**	**2**	**6**	**5**	**7**
	1	**2**	**0**	**7**	**3**	**6**
	2	**6**	**7**	**0**	**6**	**1**
	3	**5**	**3**	**6**	**0**	**5**
	4	**7**	**6**	**1**	**5**	**0**
dist(u,v)						



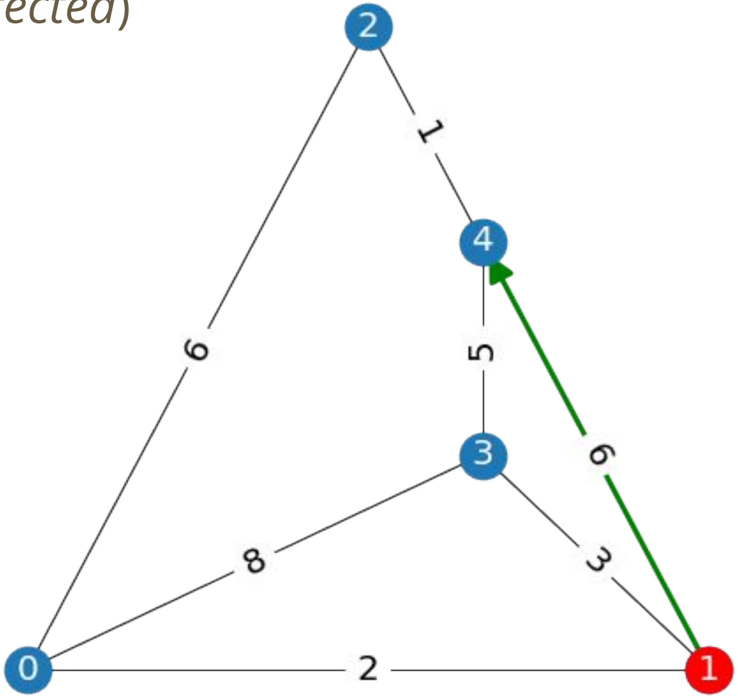
Ambil *edge* 1 → 4 (*beda*!).

Kita ingin mencari *shortest path* u ⇒ 4 mana saja yang menggunakan *edge* 1 → 4.

All-Pairs Shortest Path -- Mencari *pred*

Contoh G = (V, E) (*undirected*)

	v					
u		0	1	2	3	4
	0	**0**	**2**	**6**	**5**	**7**
	1	**2**	**0**	**7**	**3**	**6**
	2	**6**	**7**	**0**	**6**	**1**
	3	**5**	**3**	**6**	**0**	**5**
	4	**7**	**6**	**1**	**5**	**0**
dist(u,v)						



Ambil *edge* 1 → 4.

Kita ingin mencari *shortest path* u ⇒ 1 mana saja yang menggunakan *edge* 1 → 4.

Shortest path 1 ⇒ 4 menggunakan *edge* 1 → 4!

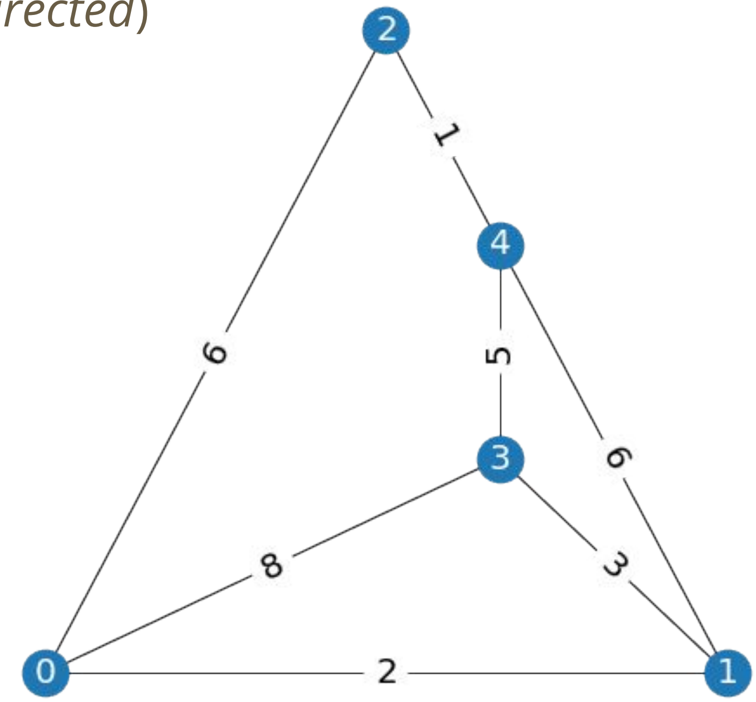
dist(1,4) = dist(1,1) + w(1 → 4)

Artinya,
pred(1,4) ← 1

All-Pairs Shortest Path -- Mencari *pred*

Contoh G = (V, E) (*undirected*)

	v					
u		0	1	2	3	4
	0	**0**	**2**	**6**	**5**	**7**
	1	**2**	**0**	**7**	**3**	**6**
	2	**6**	**7**	**0**	**6**	**1**
	3	**5**	**3**	**6**	**0**	**5**
	4	**7**	**6**	**1**	**5**	**0**
dist(u,v)						



...dan seterusnya untuk setiap *edge*.

All-Pairs Shortest Path -- Mencari *pred*

Bagaimana jika dist(u, v) = ∞ atau dist(u, v) = -∞ ?

- 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 $pred(u, u) = \text{NULL}$.

Artinya, pasti tidak ada *node* x sehingga dist(u, v) = dist(u, x) + w(x → v) sehingga pred(u,v) = NULL.

Algoritma 11: Algoritma untuk mencari nilai *pred*

masukan: graf *weighted* $G = (V, E)$, fungsi *weight* $w(u \to v)$, dan $dist[\cdot][\cdot]$ untuk graf G

keluaran: $pred[u][v]$ = *node* yang dilewati tepat sebelum *node* v pada *shortest path* $u \Rightarrow v$, untuk setiap pasang *node* $u, v \in V$

```
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
```

Mencari *pred* -- Efisiensi

Kompleksitas waktu total = **O(|V|³)**

- Inisialisasi *pred[u][v]* = NULL
  - → **O(|V| x |V|) = O(|V|²)**
- Mengisi tabel *pred*
  - → **O(|E| x |V|) = O(|V|³)**

Kompleksitas memori total = **O(|V|²)**

- Untuk menyimpan *pred[u][v]* (jawaban)
  - → **O(|V| x |V|) = O(|V|²)**

Algoritma 11: Algoritma untuk mencari nilai *pred*

masukan: graf *weighted* $G = (V, E)$, fungsi *weight* $w(u \to v)$, dan $dist[\cdot][\cdot]$ untuk graf G

keluaran: $pred[u][v]$ = *node* yang dilewati tepat sebelum *node* v pada *shortest path* $u \Rightarrow v$, untuk setiap pasang *node* $u, v \in V$

```
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
```

All-Pairs Shortest Path -- Kesimpulan

- Semua algoritma sebelumnya hanya bisa dijalankan apabila *graf* tidak mempunyai *negative cycle*!
  - Bisa dicek terlebih dahulu menggunakan Bellman-Ford (cek algoritma Johnson)
  - Ada modifikasi dari Floyd-Warshall yang bisa menangani *negative cycle*

- Apapun algoritmanya, kita bisa mencari *pred* apabila kita sudah mengetahui nilai *dist* untuk setiap pasang *node*.

All-Pairs Shortest Path -- Kesimpulan

- Kompleksitas waktu setiap algoritma...
  - Johnson → **O(|V| x |E| log |V|) = O(|V|^3 log |V|)**
  - DP (Shimbel) → **O(|V|^4)**
  - DP + DnC (Fischer-Meyer, Leyzorek) → **O(|V|^3 log |V|)**
  - DP (Floyd-Warshall) → **O(|V|^3)**

- Apakah algoritma Floyd-Warshall selalu lebih baik? **Belum tentu!**
  - Algoritma Johnson akan lebih cepat dari Floyd-Warshall pada graf yang bersifat *sparse*.
    - *sparse*: $|E| \ll |V|^2$
  - Algoritma Floyd-Warshall tidak bergantung dengan |E| sehingga bekerja dengan performa yang sama pada graf yang bersifat *sparse*.

Q&A

Referensi

1. Erickson, J. (2019). Algorithms.
2. Matrix Multiplication is Associative. Diakses 22 November 2019, https://proofwiki.org/wiki/Matrix_Multiplication_is_Associative