

IKI10400 • Struktur Data & Algoritma: *Graph*

Fakultas Ilmu Komputer • Universitas Indonesia

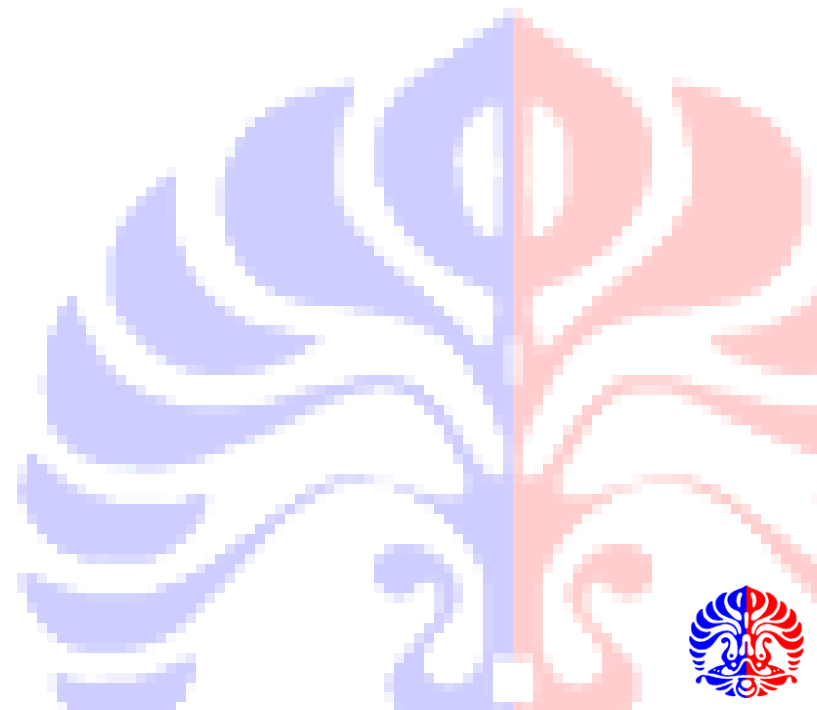
Slide acknowledgments:

Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung, Tisha Melia,
Clara Vania



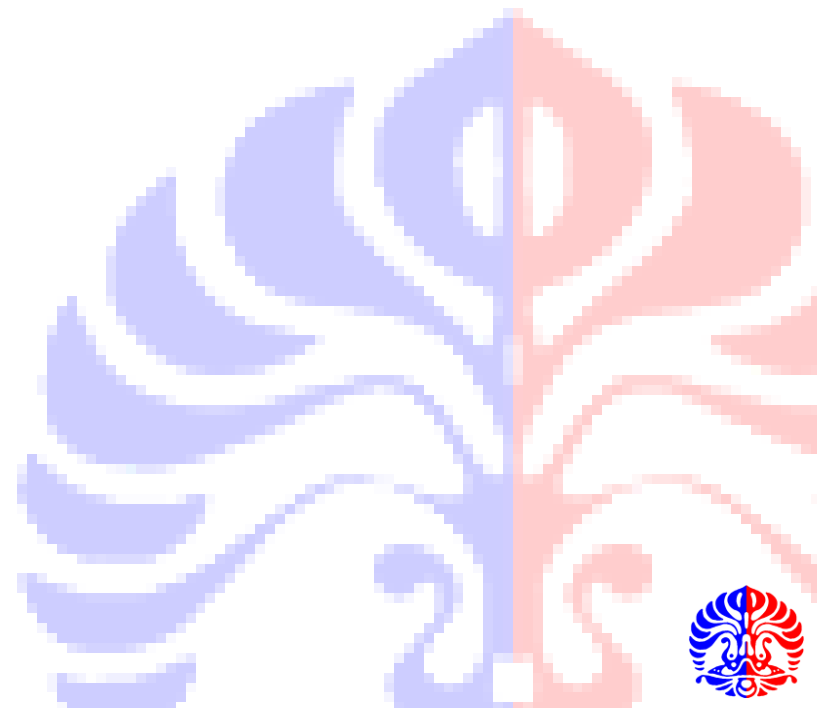
Materi

- Motivasi
- Definisi dan Istilah
- Representasi Graph
- Algoritma mencari shortest path
- Topological Sort
- Minimum spanning tree
 - Prim's Algoritma
 - Kruskal's Algoritma



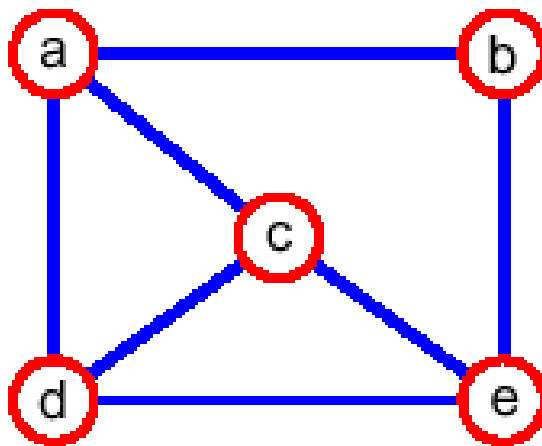
Penggunaan Graph

- Jaringan
- Peta
 - Mencari jalur terpendek
- Penjadwalan (Perencanaan Proyek)



Definisi

- Sebuah graph $G = (V, E)$ terdiri dari:
 - V : kumpulan simpul (*vertices/nodes*)
 - E : kumpulan *sisi/busur* (*edge*) yang menghubungkan simpul-simpul.
- Sebuah sisi $e = (a, b)$ memiliki informasi dua simpul yang dihubungkannya.



$V = \{a, b, c, d, e\}$

$E =$

$\{(a, b), (a, c), (a, d),$
 $(b, e), (c, d), (c, e),$
 $(d, e)\}$

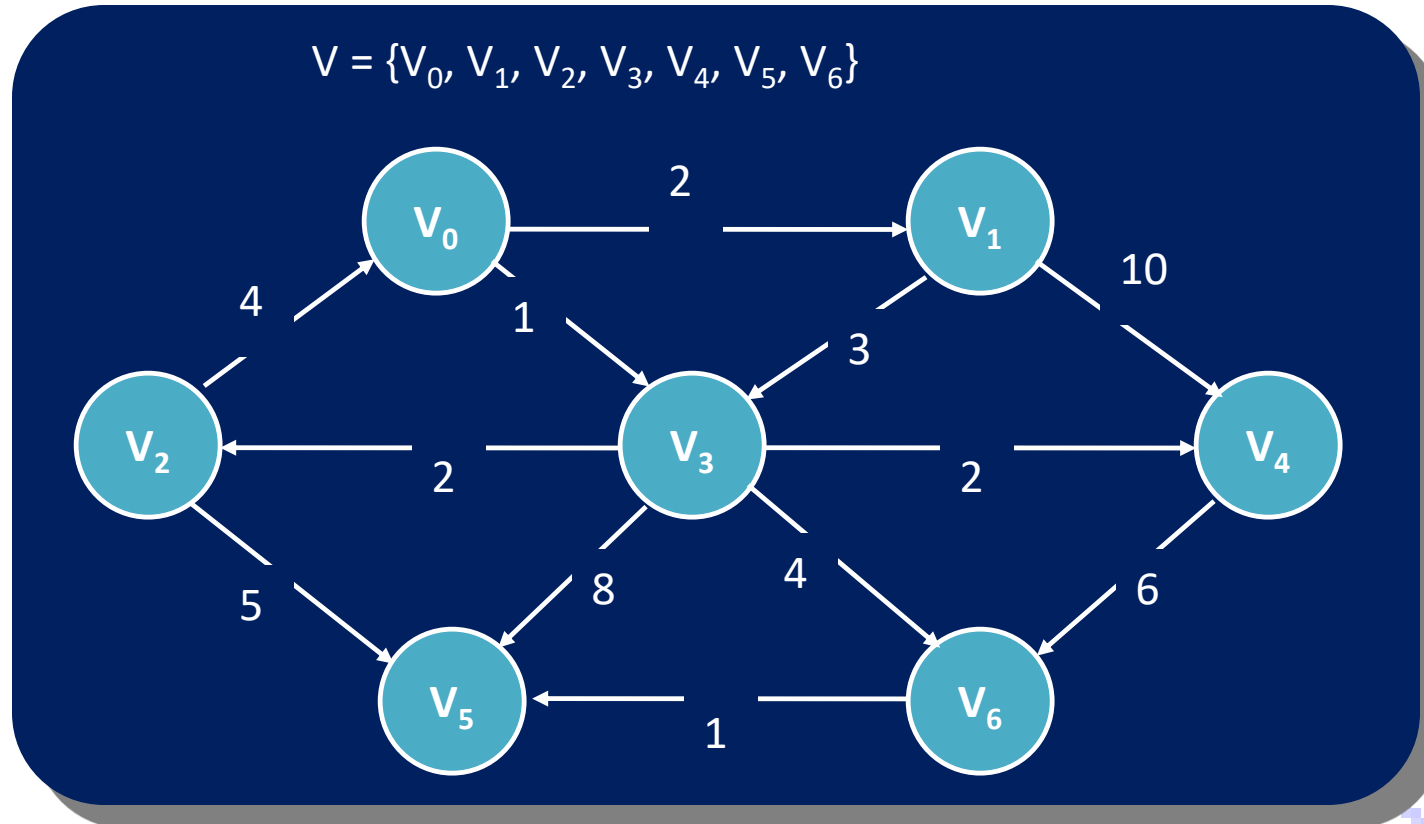
Istilah

- **undirected graph**
- **directed graph**
- **adjacent vertices**: adalah simpul-simpul yang dihubungkan oleh sebuah sisi (*edge*)
- **degree** (of a vertex): adalah jumlah simpul lain yang terhubung langsung melalui sebuah sisi.
 - Untuk kategori directed graph
 - in-degree
 - out-degree



Weighted Graph

- **weighted graph**: setiap sisi memiliki **bobot/nilai**.



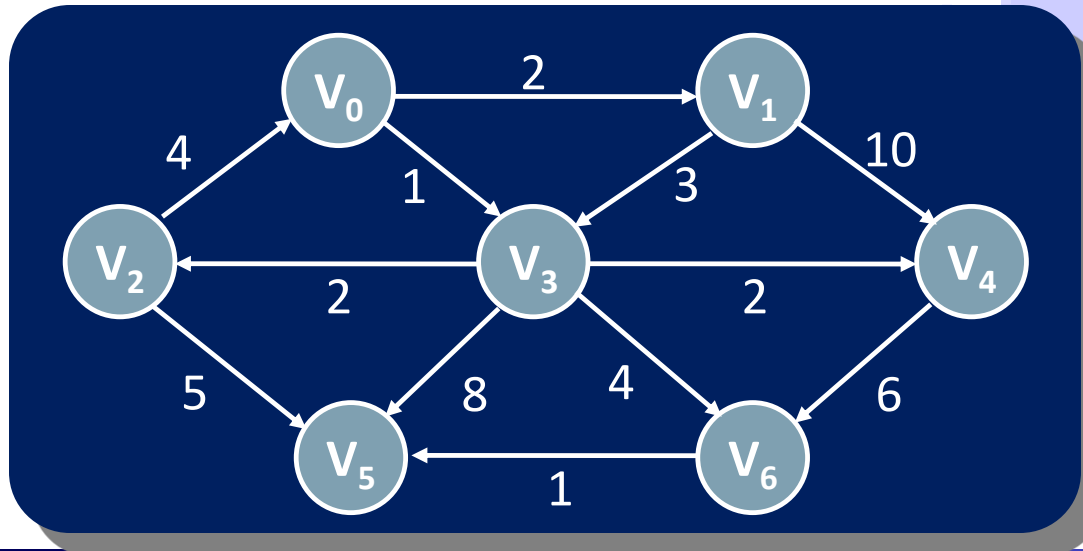
$(V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10)$
 $(V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2)$
 $(V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1)$

■ $|V| = 7; |E| = 12$



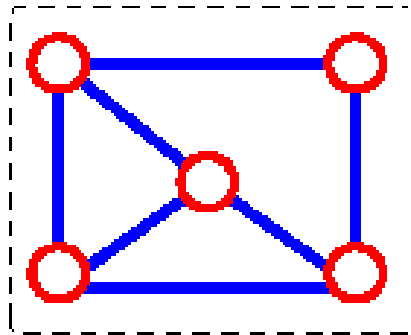
Istilah

- **Jalur/path**: urutan simpul (vertices) v_1, v_2, \dots, v_k sedemikian sehingga simpul yang berurutan v_i dan v_{i+1} adalah simpul yang terhubung.
- **simple path**: tidak ada simpul yang diulang.
- **cycle**: simple path, dengan catatan simpul awal sama dengan simpul akhir
- **DAG (Directed Acyclic Graph)**: Graph dengan busur/sisi yang memiliki arah dan tidak memiliki cycles.

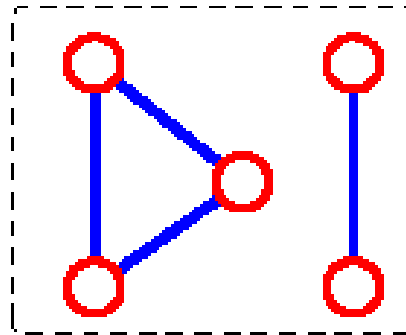


Istilah

- *connected graph*: tiap simpul terhubung dengan simpul lain



connected

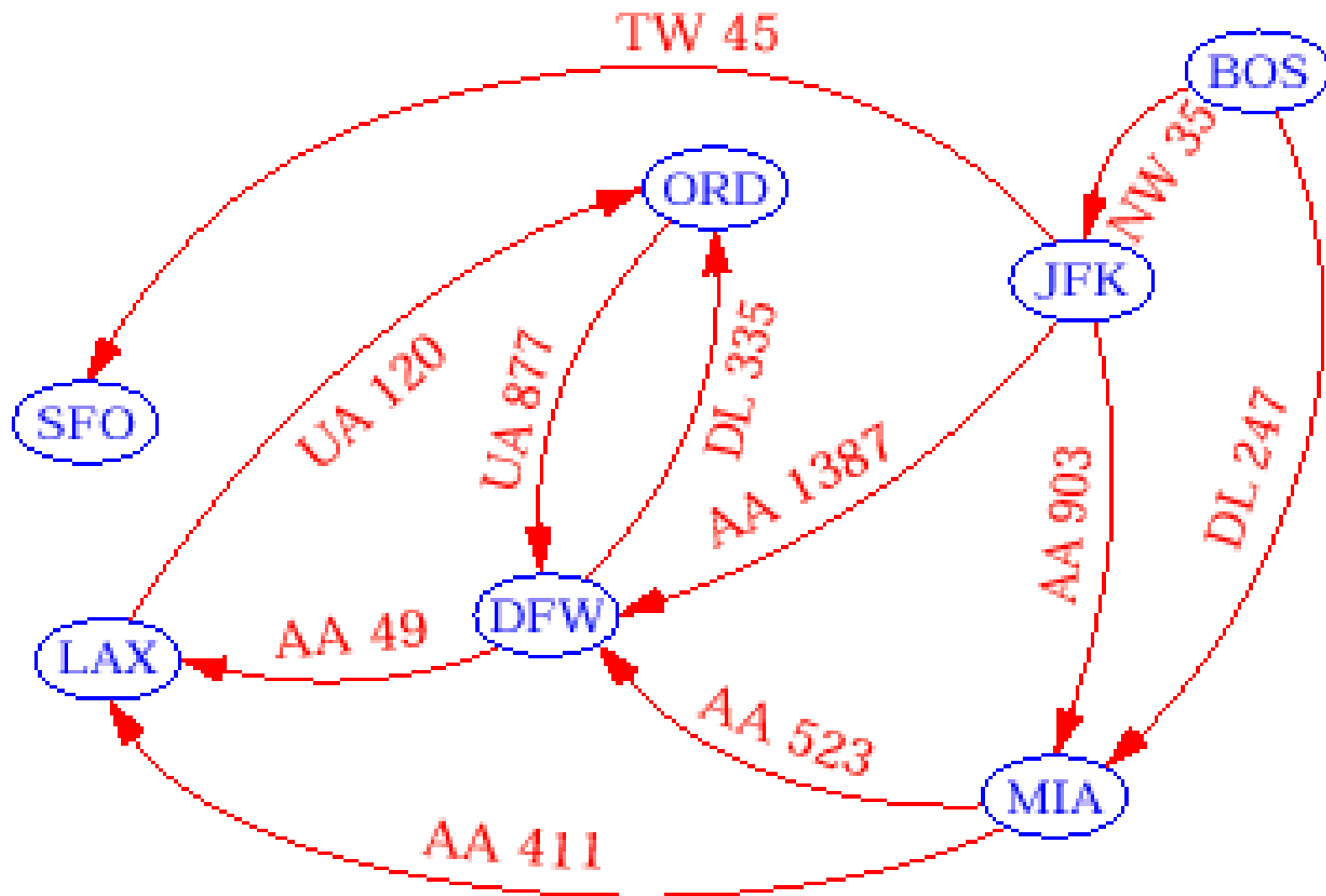


not connected

- *subgraph*: bagian simpul dan sisi yang dapat membentuk graph

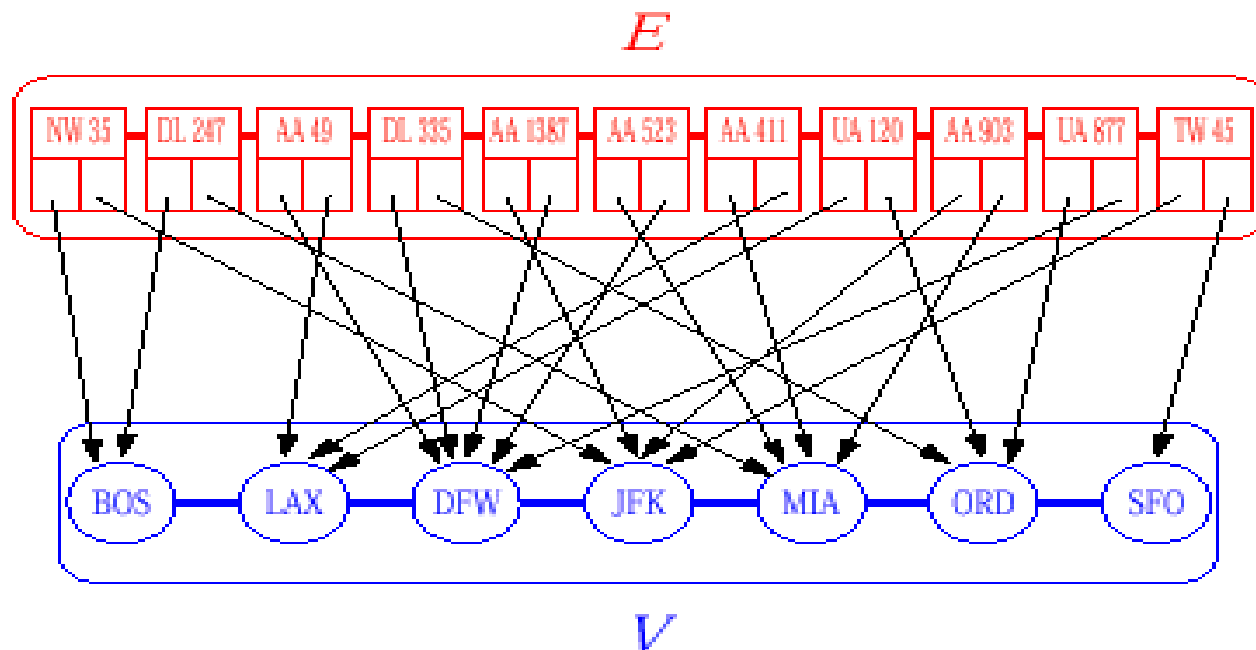


Representasi



Representasi: *Edge List*

- Struktur **edge list** hanya menyimpan simpul dan sisi dalam sebuah list yang tidak terurut.
- Pada tiap sisi disimpan informasi simpul yang terhubung oleh sisi tersebut.
- mudah diimplementasikan.
- Tidak efisien dalam keperluan mencari sisi bila diketahui simpulnya.



Edge List: Representation

```
class Node
{
    String label;
}
```

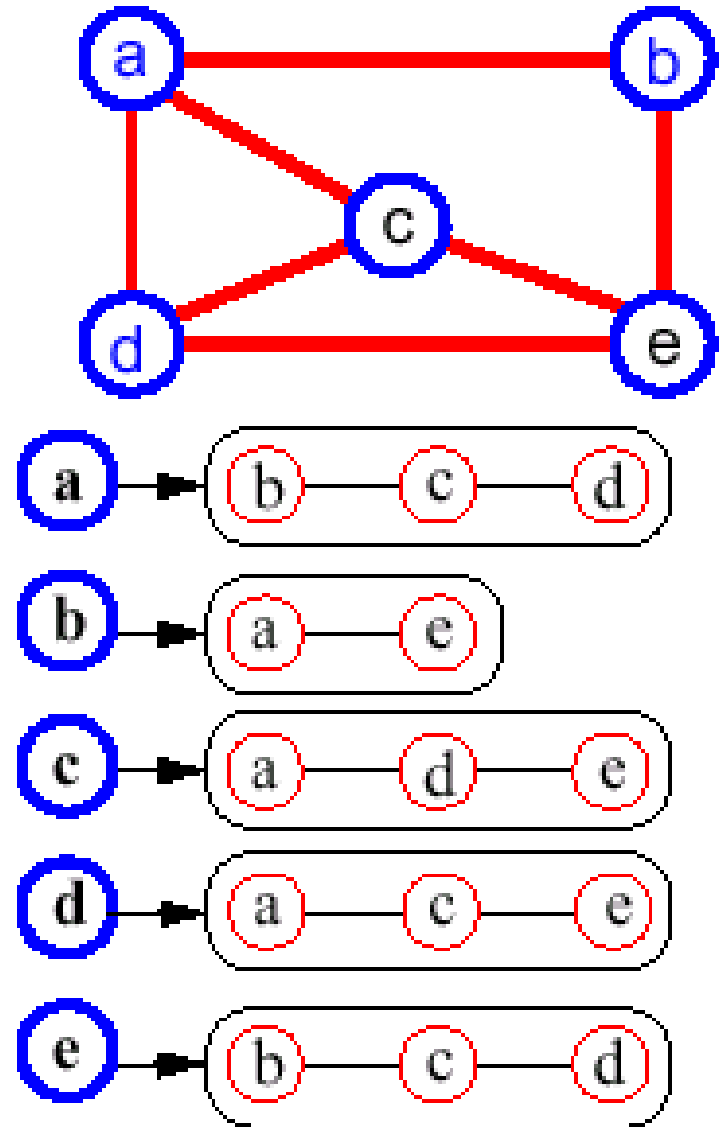
```
class Edge
{
    Node from;
    Node to;
    String label;
    int weight;
}
```

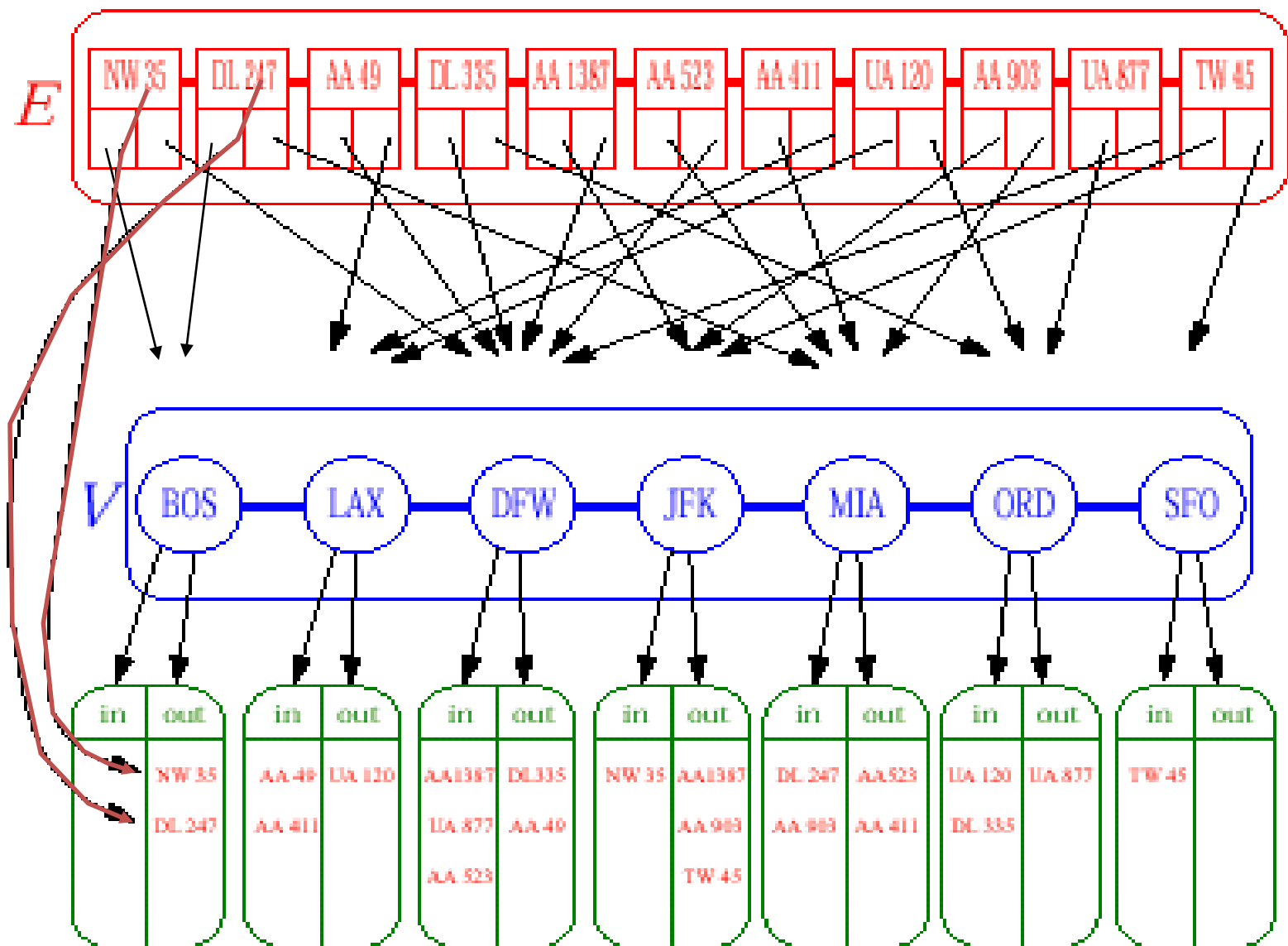
```
class Graph
{
    List<Edge> edgeList;
}
```



Representasi: *Adjacency List* (traditional)

- **Adjacency list** dari sebuah *vertex* v adalah sekumpulan *vertex* yang terhubung dengan v
- Merepresentasikan graph, dengan menyimpan daftar *adjacency lists* dari seluruh *vertex*.
- struktur **adjacency list** dapat digabungkan dengan struktur edge list.





Adjacency List: Representation

```
class Node
{
    String label;
}
```

```
class Edge
{
    Node from;
    Node to;
    String label;
    int weight;
}
```

```
class AdjacencyList
{
    Node node;
    List<Edge> adjacent;
}
```

```
class Graph
{
    List<AdjacencyList> adjacencyLists;
}
```



Adjacency List: Representation (alt.)

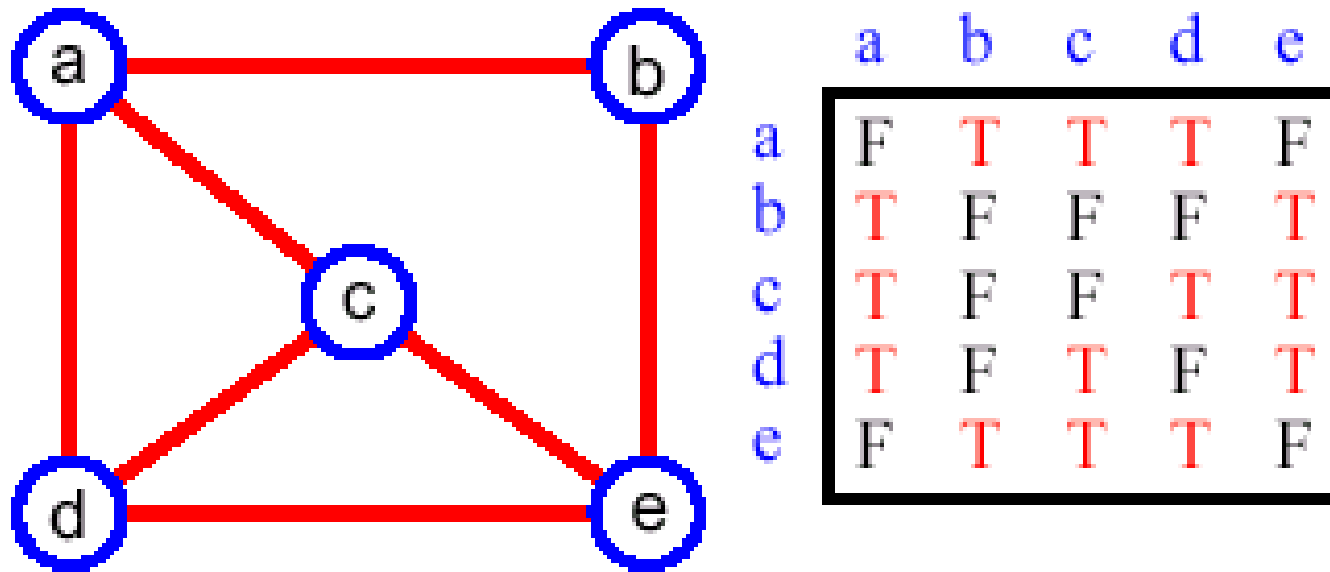
```
class Node
{
    String label;
}
```

```
class Edge
{
    Node from;
    Node to;
    String label;
    int weight;
}
```

```
class Graph
{
    Map<Node, List<Edge>> adjacencyLists;
}
```



Representasi: Adjacency Matrix (traditional)



- matrix M dengan elemen setiap pasang simpul
 - $M[i,j] = \text{true}$ artinya ada sisi dari simpul (i,j) di graph.
 - $M[i,j] = \text{false}$ artinya tidak ada sisi dari simpul (i,j) di graph.



Representation: Adjacency Matrix

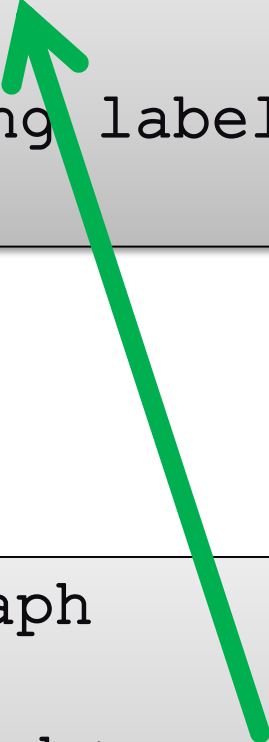
	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

BOS DFW JFK LAX MIA ORD SFO
0 1 2 3 4 5 6

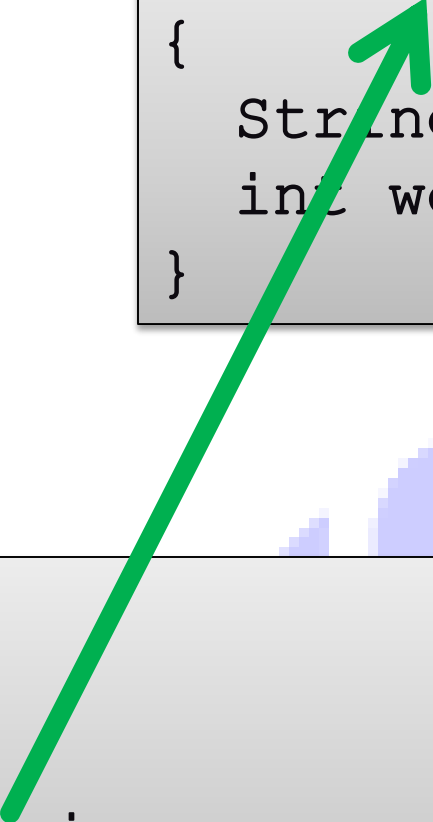


Representation: Adjacency Matrix

```
class Node
{
    String label;
}
```



```
class Edge
{
    String label;
    int weight;
}
```



```
class Graph
{
    List<Node> nodeList;
    Edge[][] adjacencyMatrix;
}
```

see the other slides

GRAPH TRAVERSAL

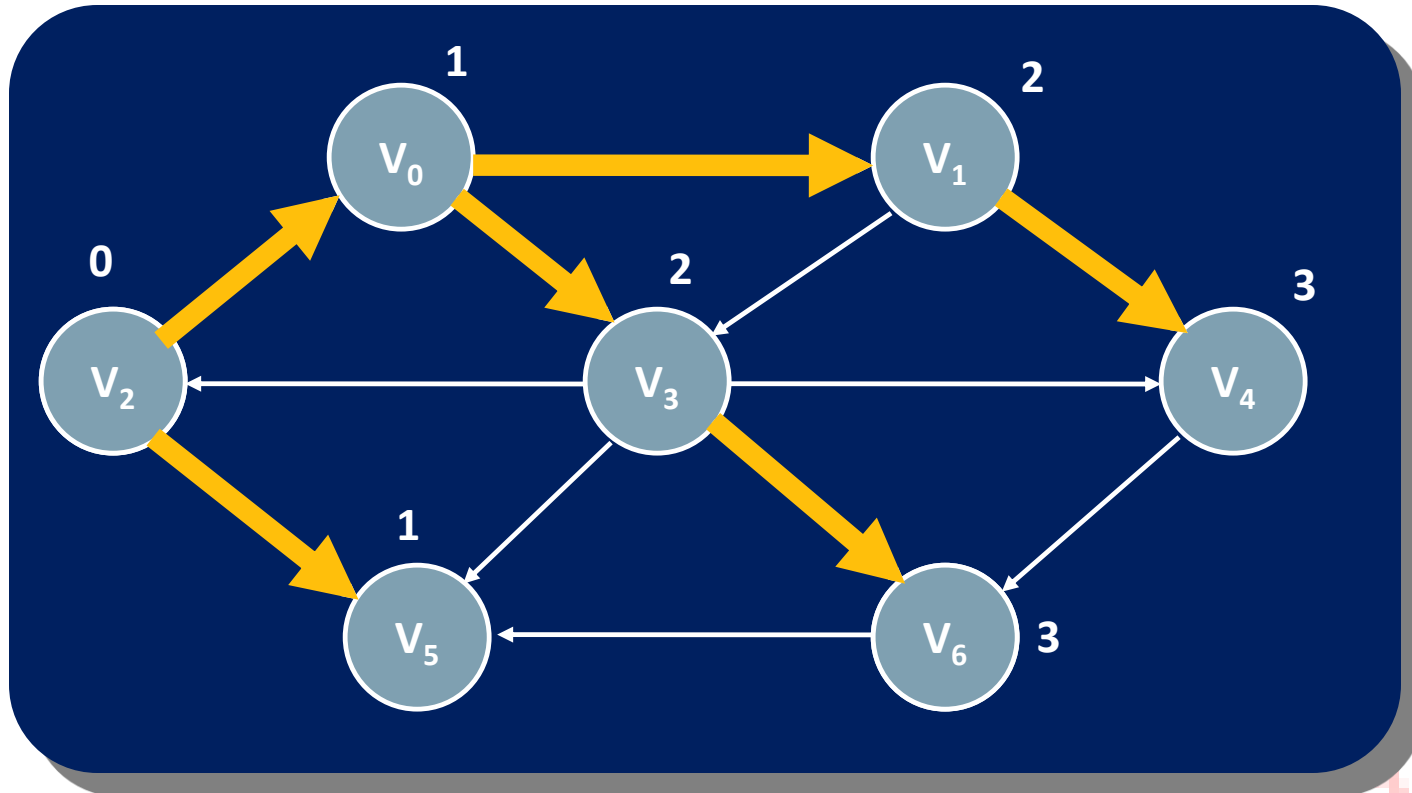


GRAPH ALGORITHMS



Shortest Path:

- Vertex awal: V_2
- Bila sisi tidak memiliki bobot, gunakan algoritma BFS (**Breadth First Search**).



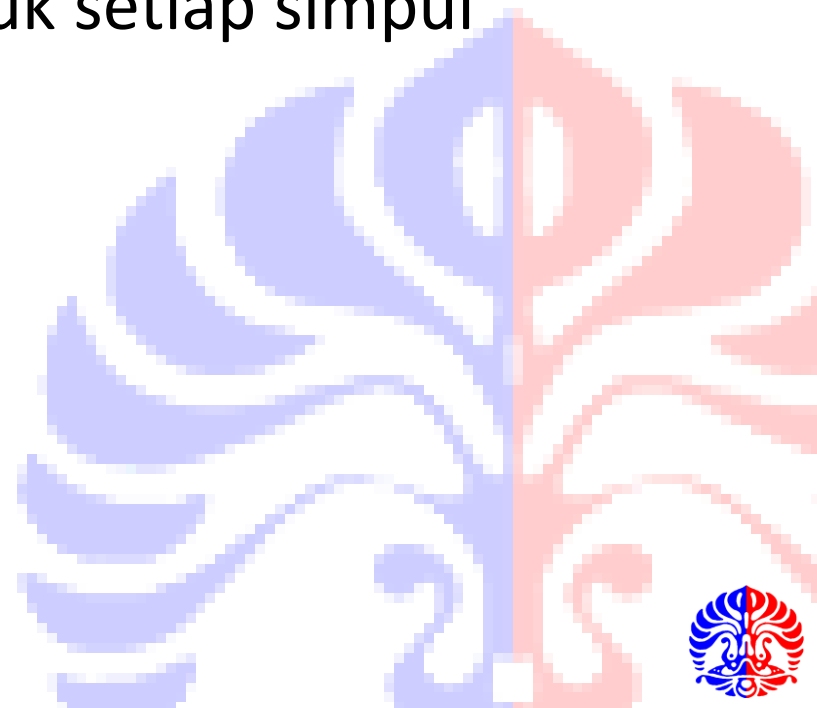
Dijkstra's Algorithm

- Banyak masalah → weighted graph (mis: jaringan transport)
- Algoritma **Dijkstra** menghitung jarak tiap simpul dari simpul awal hingga akhirnya diketahui jarak terpendek simpul akhir yang diinginkan.
- Algoritma mengingat simpul mana saja yang telah dihitung jarak terpendeknya dan dinyatakan dalam kelompok hijau (pada literatur dinyatakan sebagai awan putih/white cloud).
- Untuk simpul yang baru sebagian dihitung jaraknya dan belum bisa dipastikan apakah itu jarak terpendek, dinyatakan dengan kelompok abu-abu.
- Untuk simpul yang sama sekali belum dihitung, dinyatakan dalam kelompok hitam.



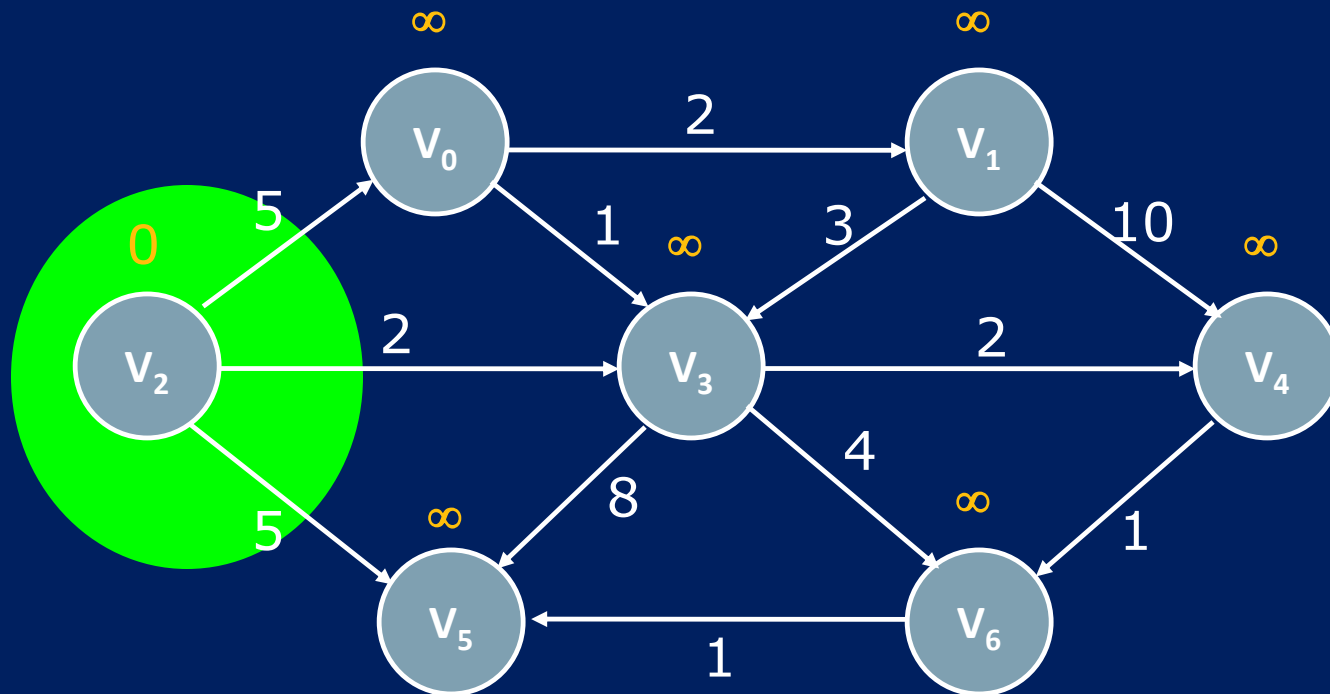
Dijkstra's Algorithm

- Algoritma menggunakan label $D[v]$ untuk menyimpan perkiraan jarak terpendek antara s dan v .
- Ketika sebuah simpul v ditambahkan kedalam kelompok abu-abu nilai $D[v]$ sama dengan bobot antara s dan v .
- pada awalnya, nilai label D untuk setiap simpul adalah:
 - $D[s] = 0$
 - $D[v] = \infty$ untuk $v \neq s$



Dijkstra's Algorithm: *stages*

- Awal: Tentukan simpul awal.



Expanding the White Cloud

- Setiap penambahan simpul, kita harus uji apakah jalur melalui u lebih baik.
- Misalkan u adalah sebuah simpul yang tidak berada di **kelompok hijau**, tapi sudah diketahui jarak terpendeknya dari s
 - tambahkan u ke dalam kelompok hijau
 - hitung jarak **simpul lain** dengan algoritma berikut:

Untuk tiap simpul z yang terhubung ke u lakukan:

jika z tidak di kelompok hijau maka

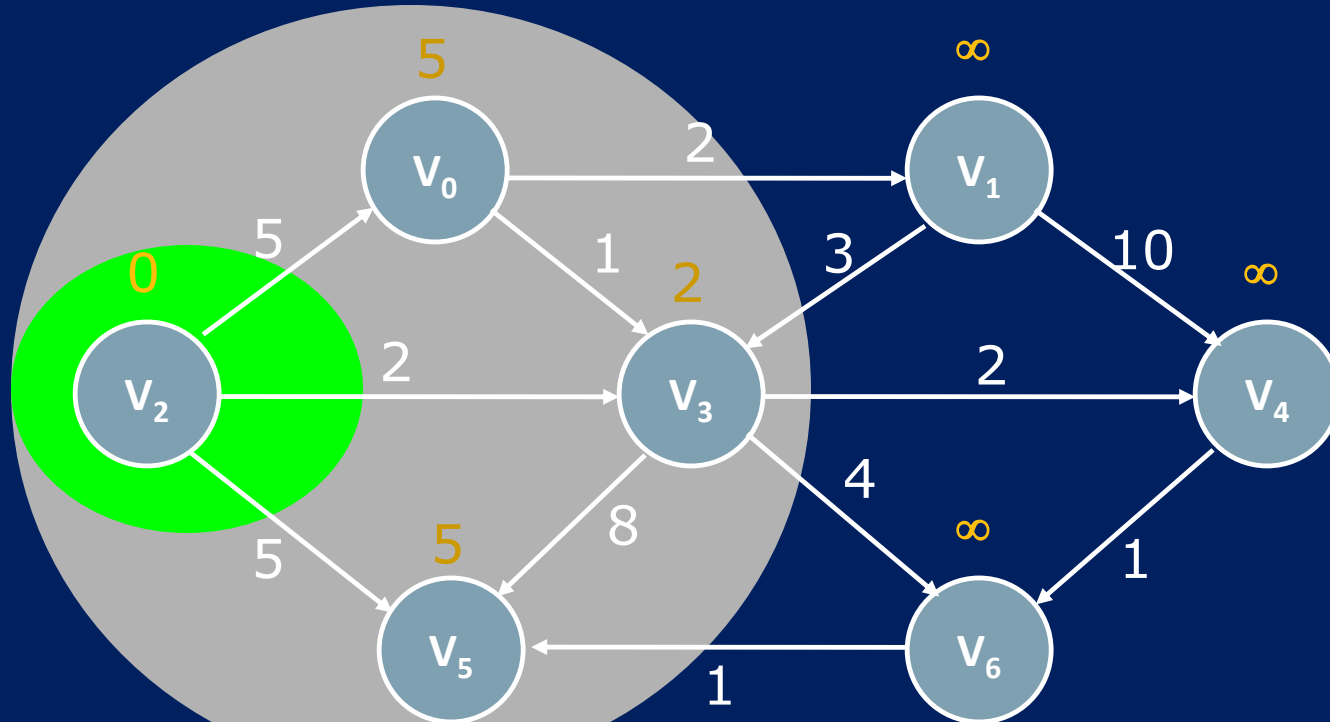
if $D[u] + \text{bobot}(u, z) < D[z]$ then

$D[z] = D[u] + \text{bobot}(u, z)$



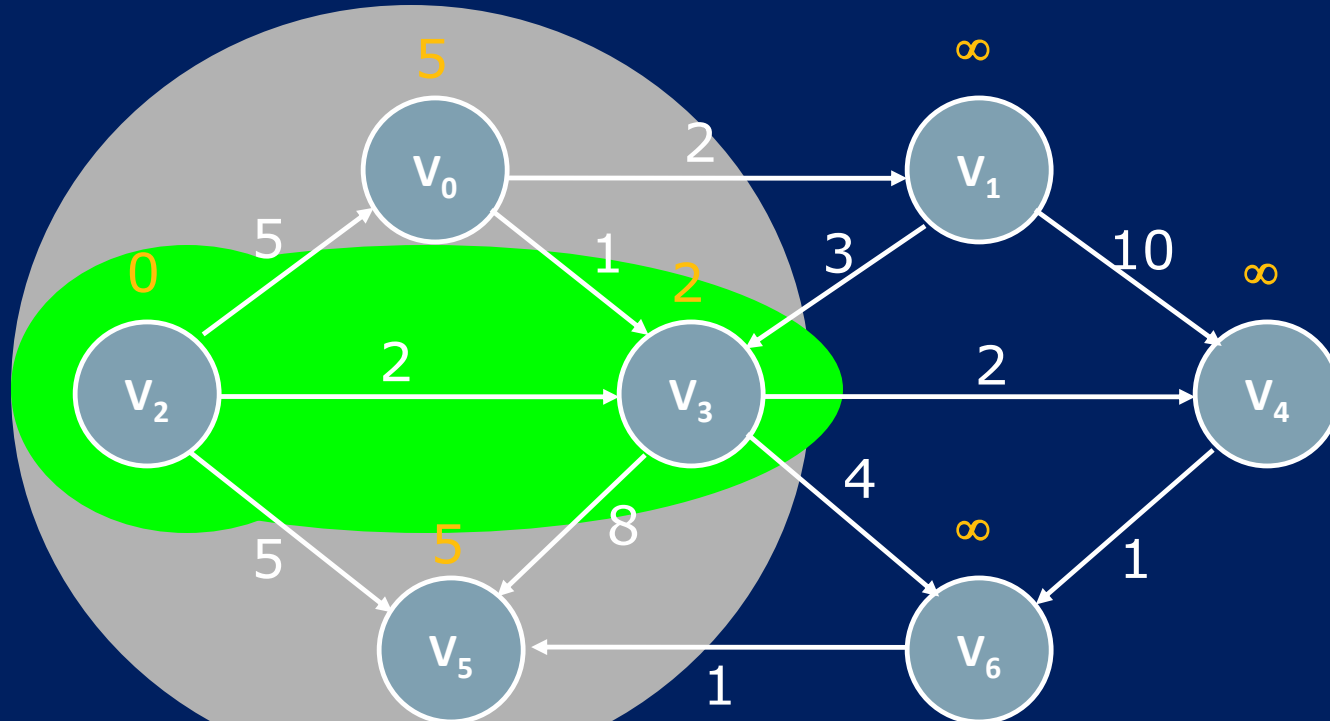
Dijkstra's Algorithm: *stages*

- setelah V_2 ditambahkan ke kelompok hijau, hitung $D[V_x]$ untuk setiap V_x yang terhubung ke V_2



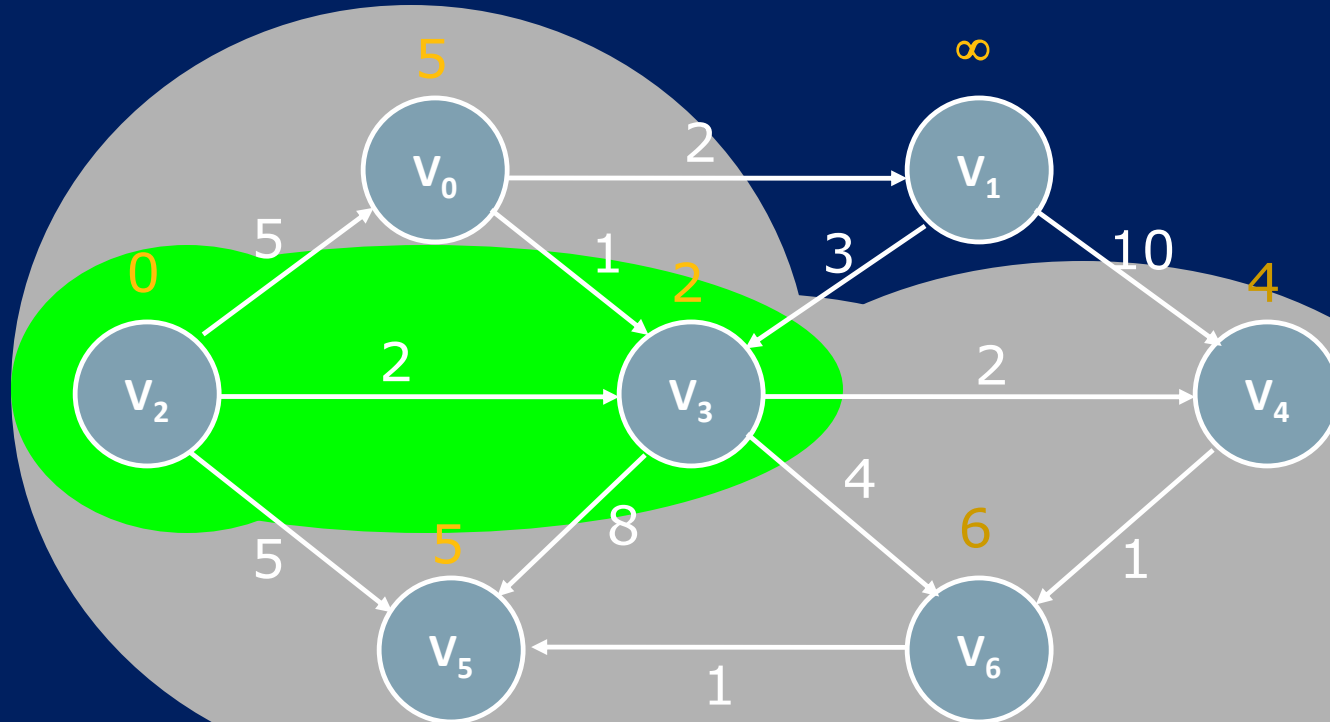
Dijkstra's Algorithm: stages

- Tambahkan ke dalam kelompok hijau simpul pada kelompok abu-abu yang memiliki nilai $D[V]$ minimum.
- Pada contoh adalah simpul V_3



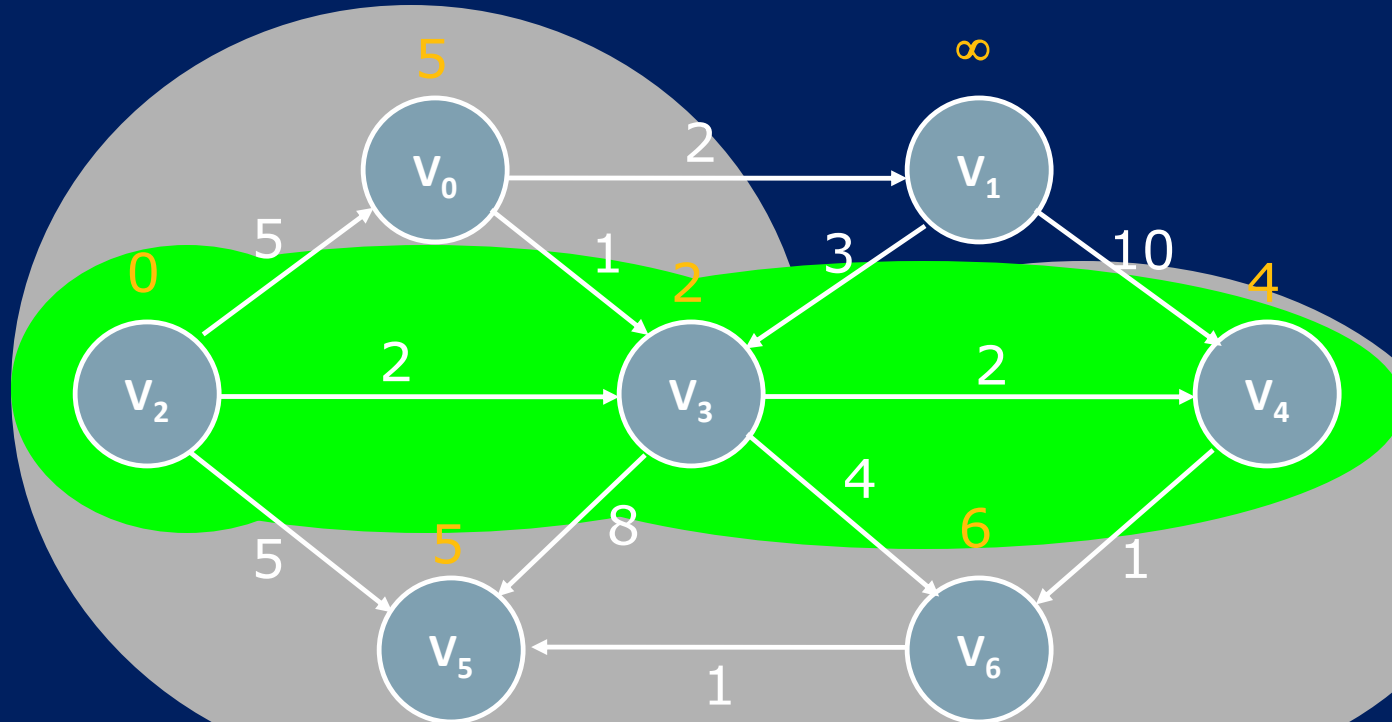
Dijkstra's Algorithm: stages

- Setelah V_3 ditambahkan ke kelompok hijau, hitung $D[V_x]$ untuk setiap V_x yang terhubung dengan V_3 . Simpul-simpul tersebut menjadi kelompok abu-abu.



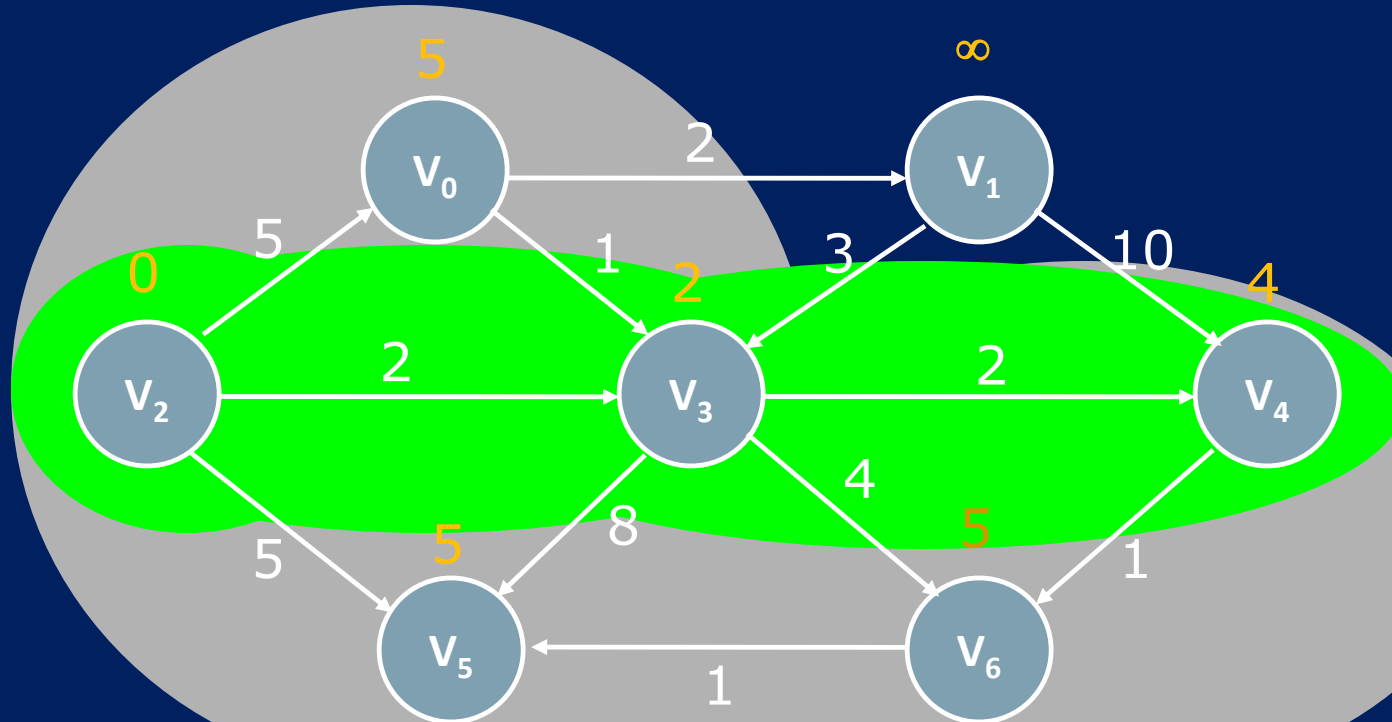
Dijkstra's Algorithm: stages

- Pilih dari kelompok abu-abu, simpul yang memiliki nilai $D[V]$ paling minimum dan tambahkan pada kelompok hijau.



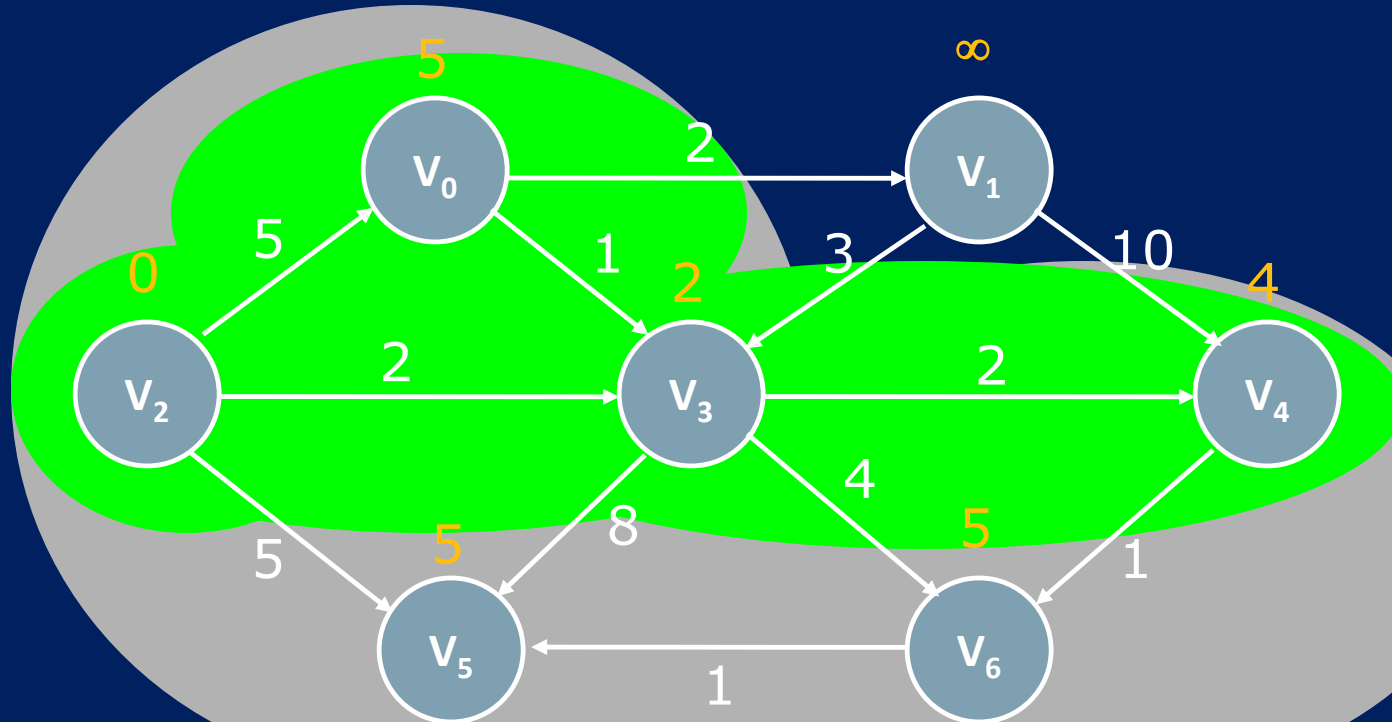
Dijkstra's Algorithm: stages

- Setelah V_4 ditambahkan ke kelompok hijau, hitung $D[V_x]$ untuk setiap V_x yang terhubung dengan V_4 . Simpul-simpul tersebut menjadi kelompok abu-abu.



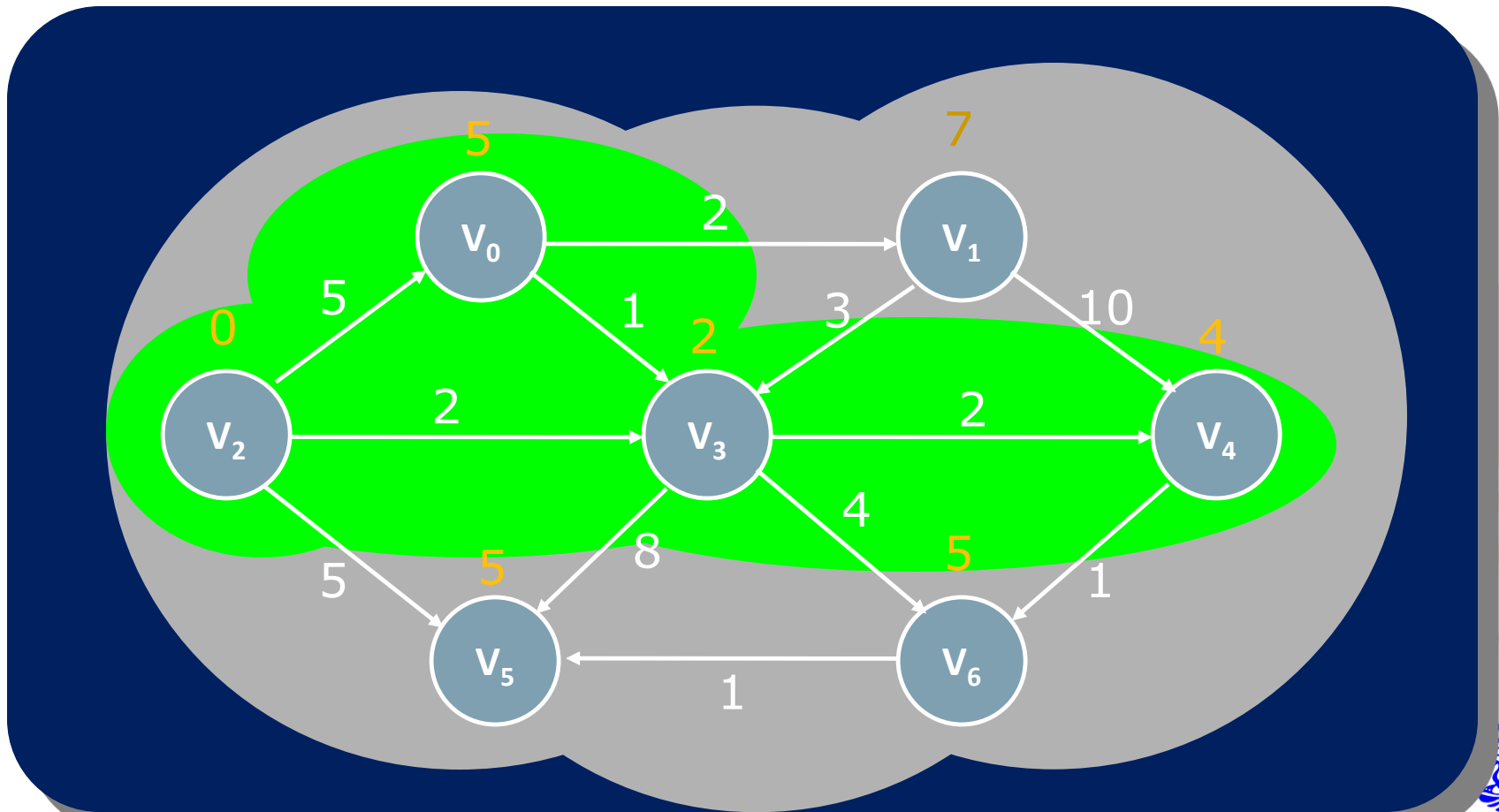
Dijkstra's Algorithm: stages

- Pilih dari kelompok abu-abu, simpul yang memiliki nilai $D[V]$ paling minimum dan tambahkan pada kelompok hijau.

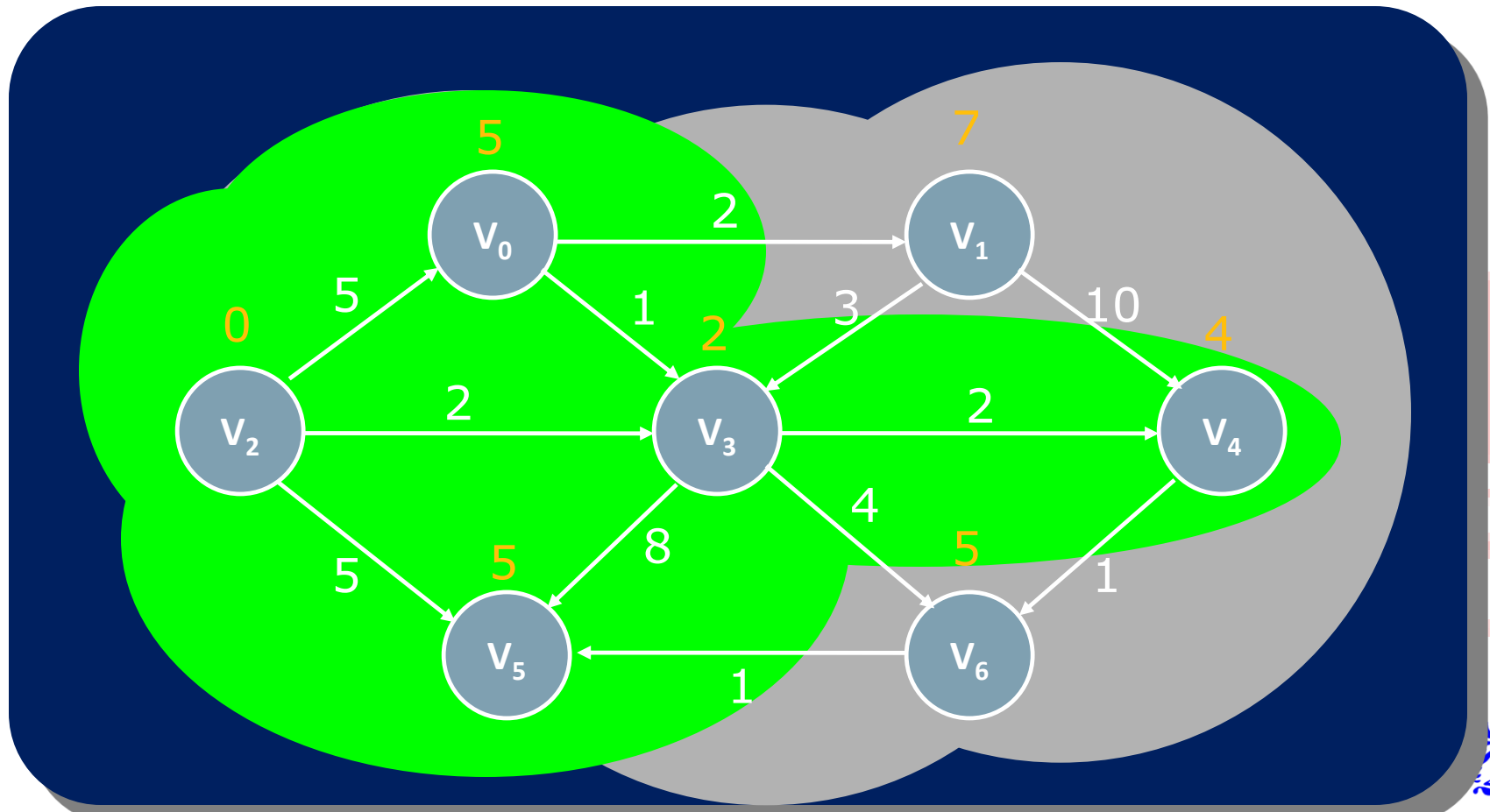


Dijkstra's Algorithm: stages

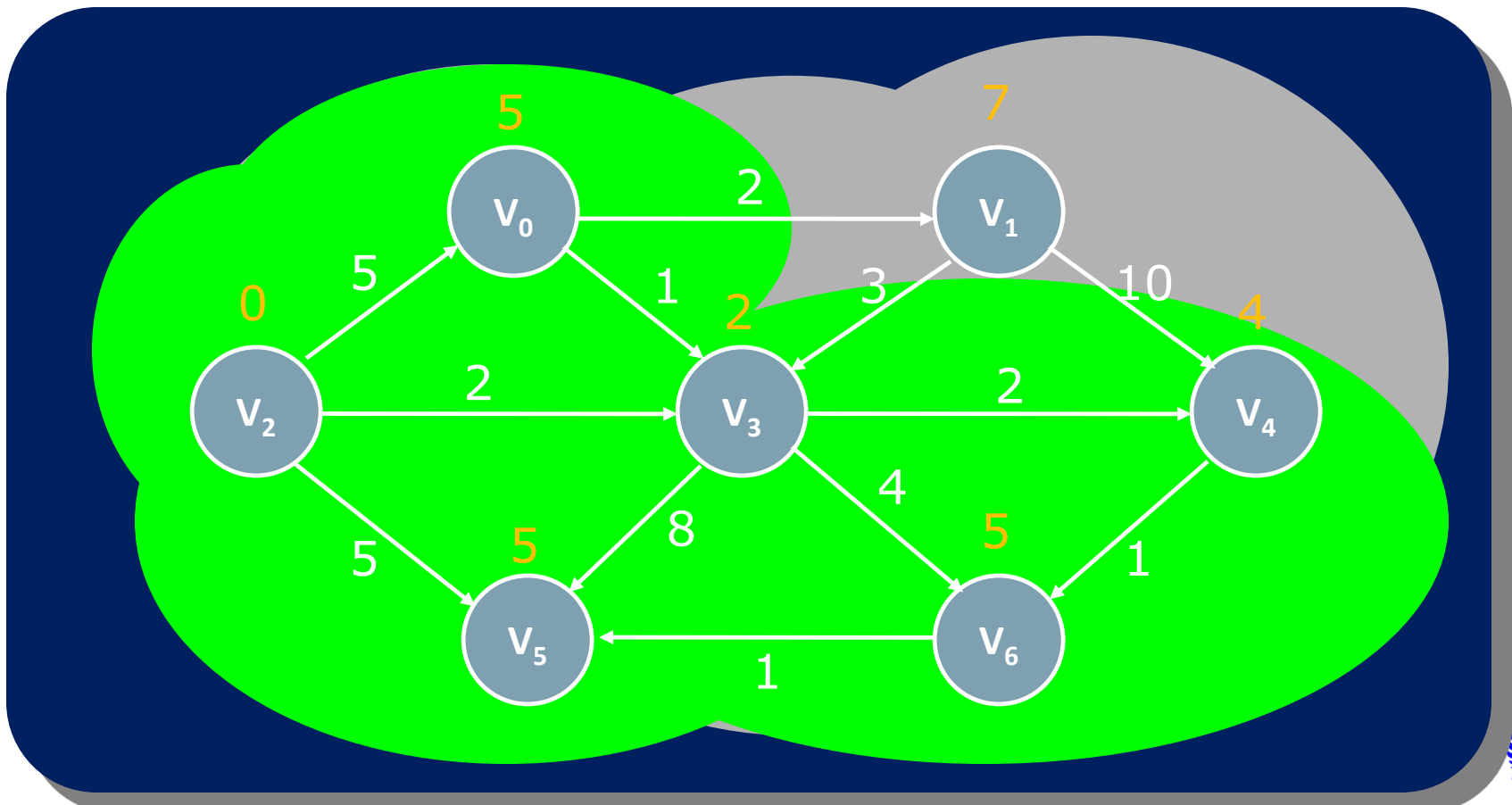
- Setelah V_4 ditambahkan ke kelompok hijau, hitung $D[V_x]$ untuk setiap V_x yang terhubung dengan V_4 . Simpul-simpul tersebut menjadi kelompok abu-abu.



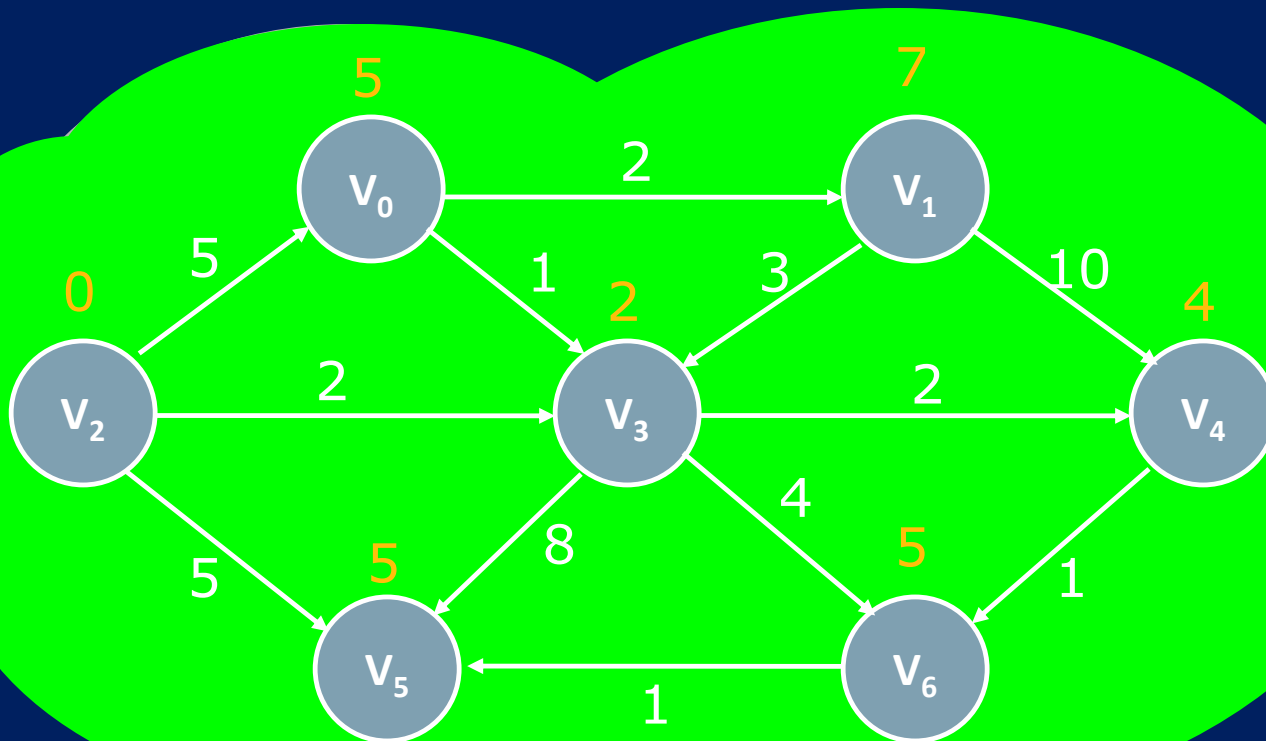
- Pilih dari kelompok abu-abu, simpul yang memiliki nilai $D[V]$ paling minimum dan tambahkan pada kelompok hijau.
- Setelah V_5 ditambahkan ke kelompok hijau, hitung $D[V_x]$ untuk setiap V_x yang terhubung dengan V_5 . Simpul-simpul tersebut menjadi kelompok abu-abu.



- Pilih dari kelompok abu-abu, simpul yang memiliki nilai $D[V]$ paling minimum dan tambahkan pada kelompok hijau.
- Setelah V_6 ditambahkan ke kelompok hijau, hitung $D[V_x]$ untuk setiap V_x yang terhubung dengan V_6 .

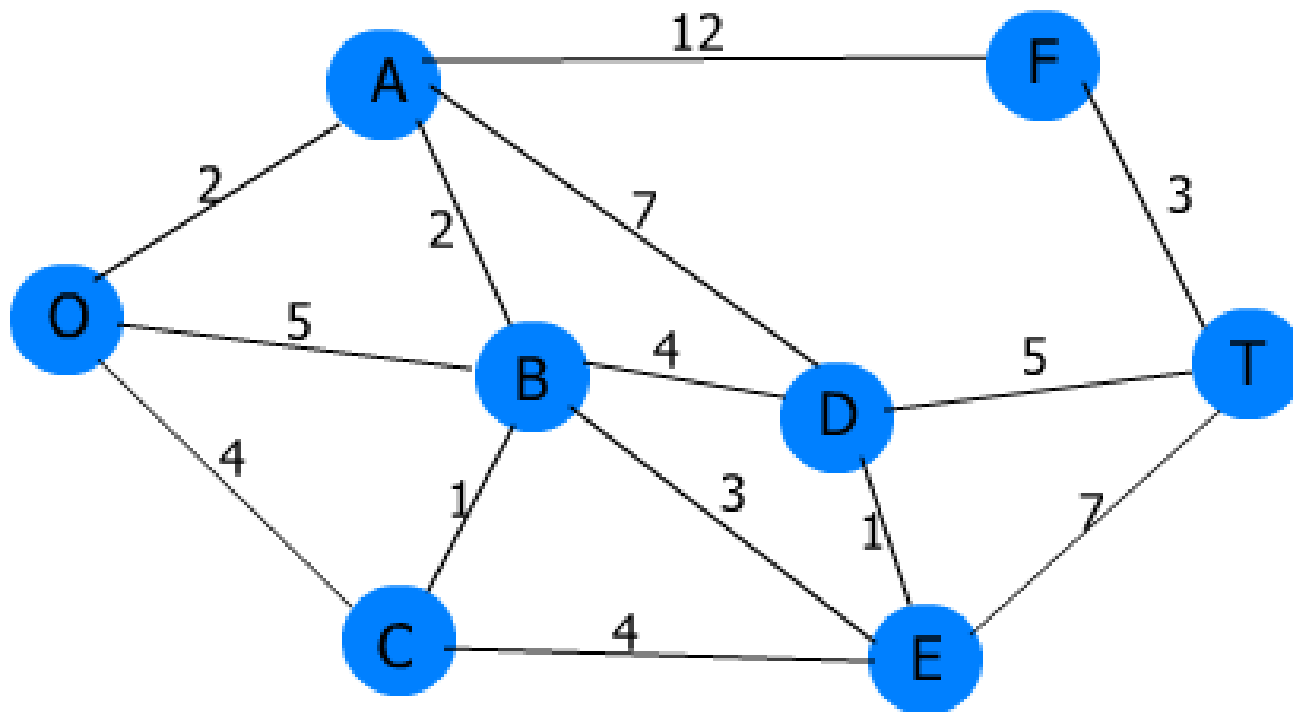


- Pilih dari kelompok abu-abu, simpul yang memiliki nilai $D[V]$ paling minimum dan tambahkan pada kelompok hijau.
- Setelah V_1 ditambahkan ke kelompok hijau, hitung $D[V_x]$ untuk setiap V_x yang terhubung dengan V_1 .

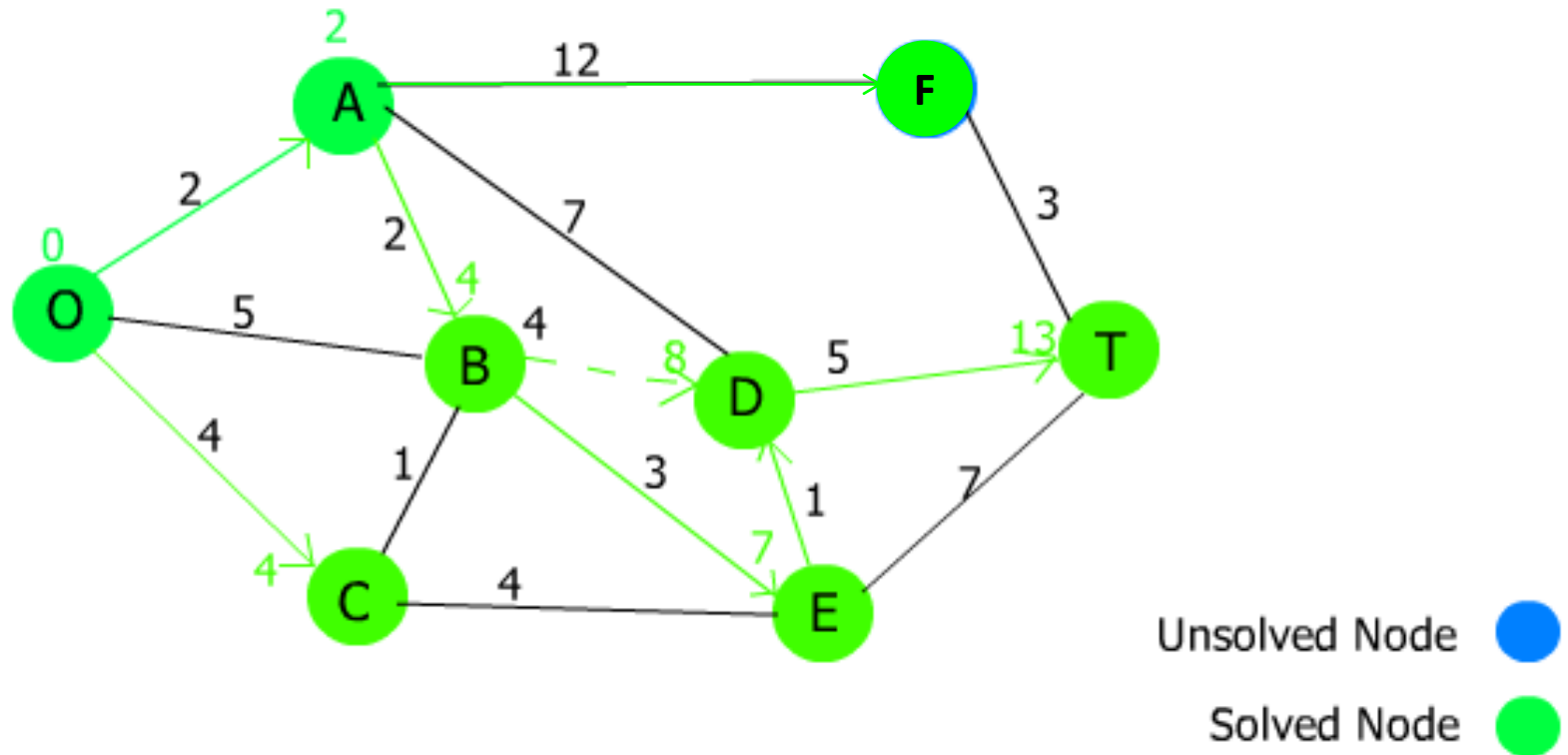


Latihan

- Tentukan jarak minimum dari verteks O ke setiap verteks pada graph!

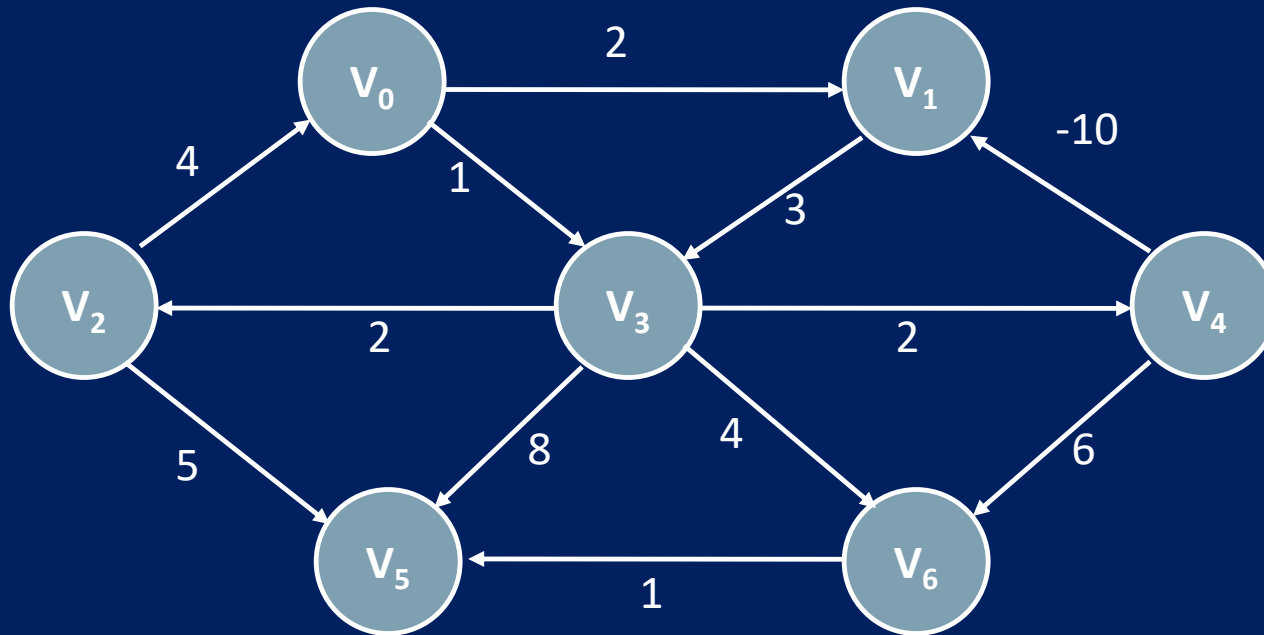


Jawaban



Variasi shortest path problem

- Negative-weighted Shortest-path



- All-Pair Shortest Path: Floyd



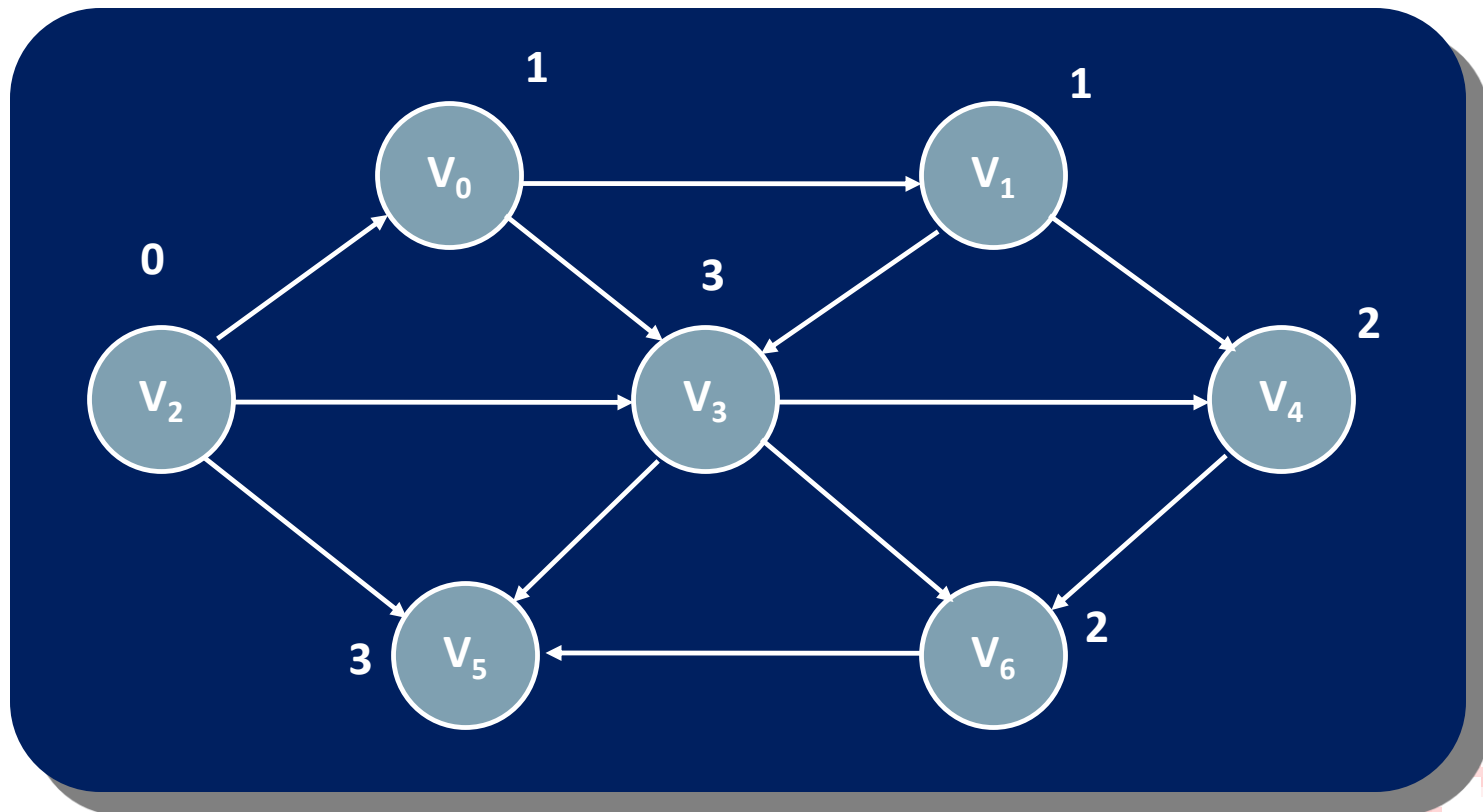
Topological Sorting

- Sebuah *topological sort* dari sebuah *directed acyclic graph* adalah sebuah urutan simpul sehingga tiap sisi/busur terurut dari kiri ke kanan (atau sebaliknya).
- Setiap DAG memiliki minimal satu topological sort.
- Gunakan algoritma *depth-first search* untuk menentukan topological sort dari sebuah DAG.
- sebuah graph yang memiliki *cycle*, tidak memiliki *topological sort*.
- Contoh permasalahan:
 - Urutan pengerjaan proyek bangunan

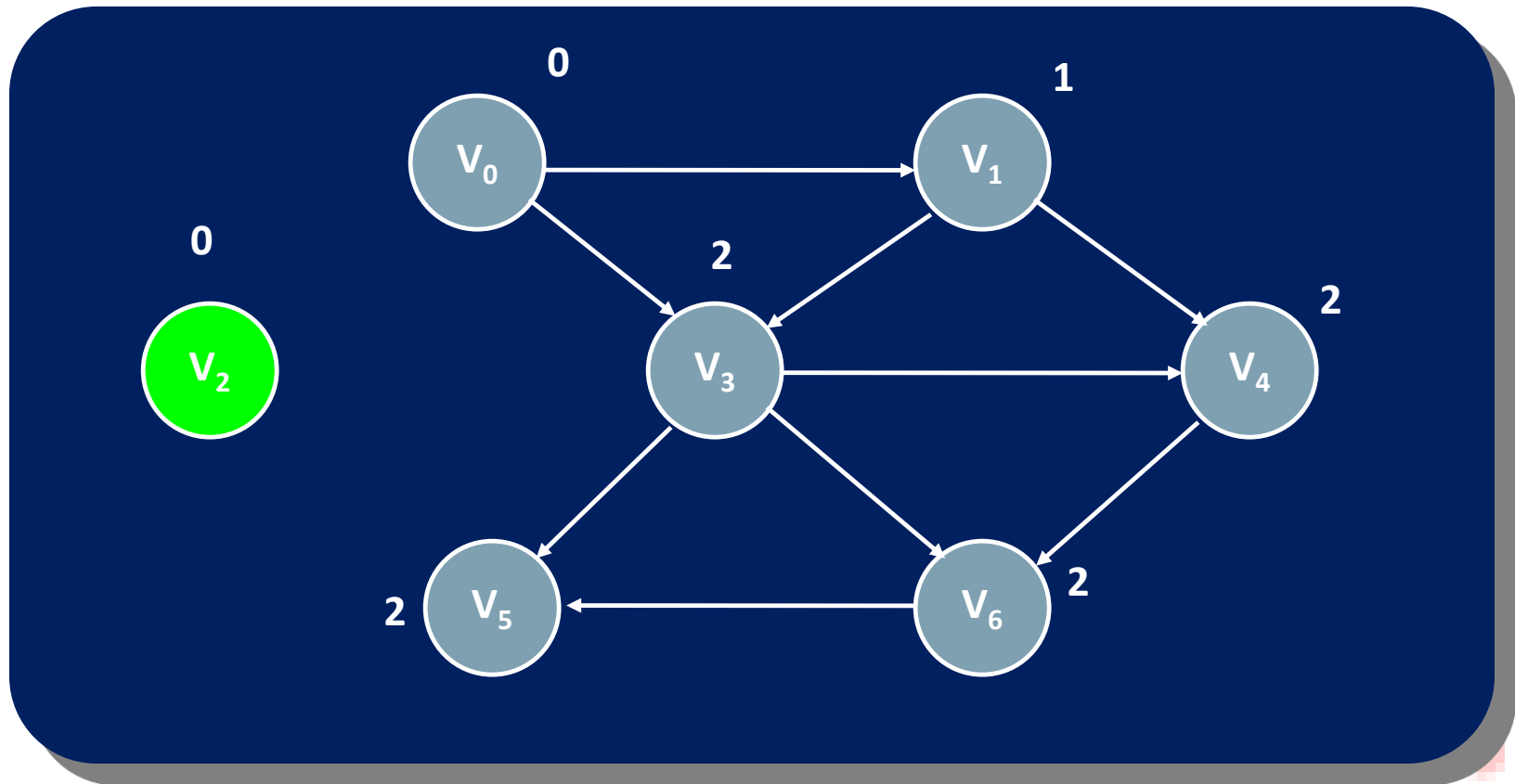


Topological Sorting: Algoritma

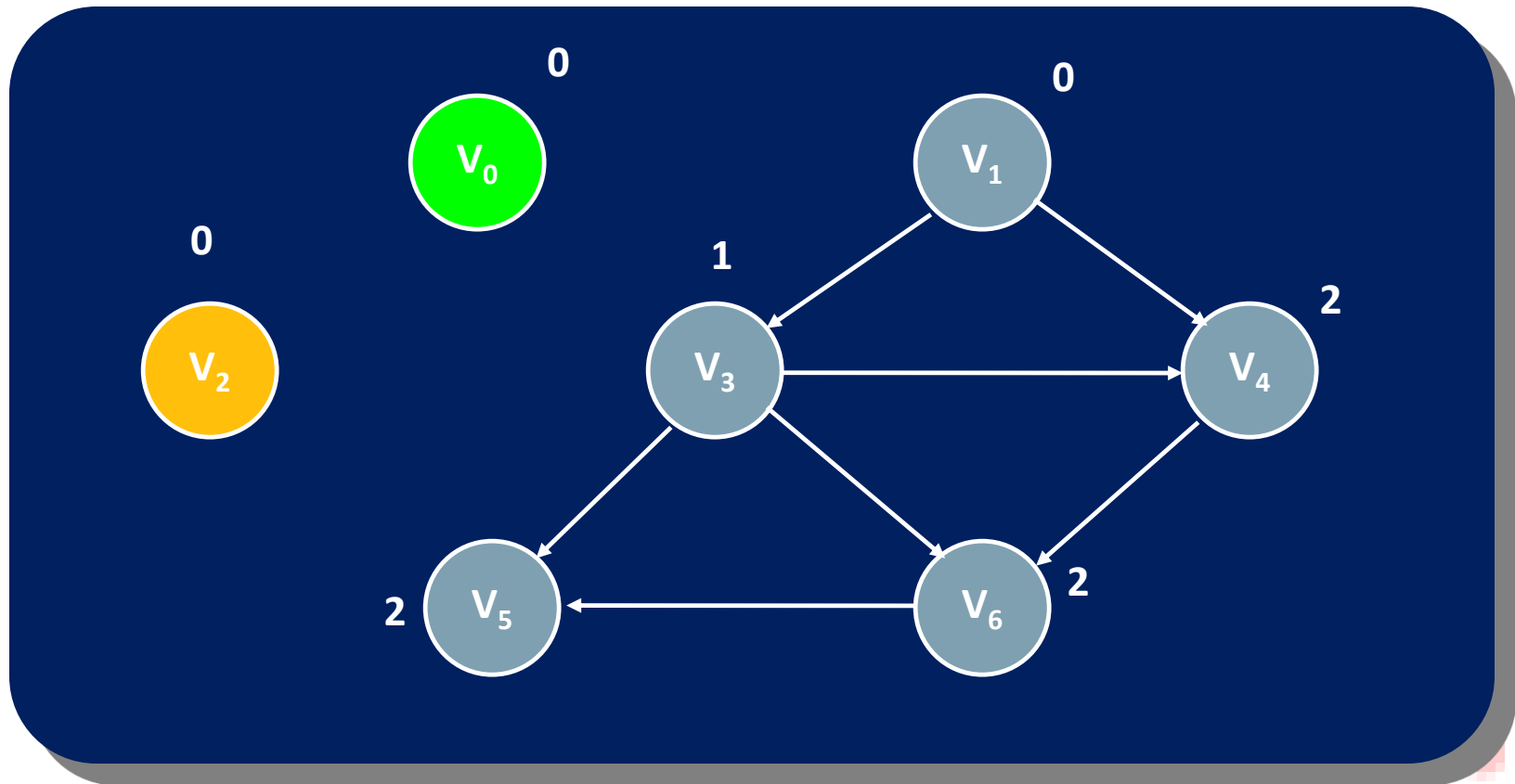
- Mulai dari sebuah simpul dengan in-degree = 0 (Tidak ada panah/sisi yang menuju simpul tersebut.)
- buang semua sisi yang berasal dari simpul tersebut.
- Sesuaikan nilai in-degree simpul lain-nya.



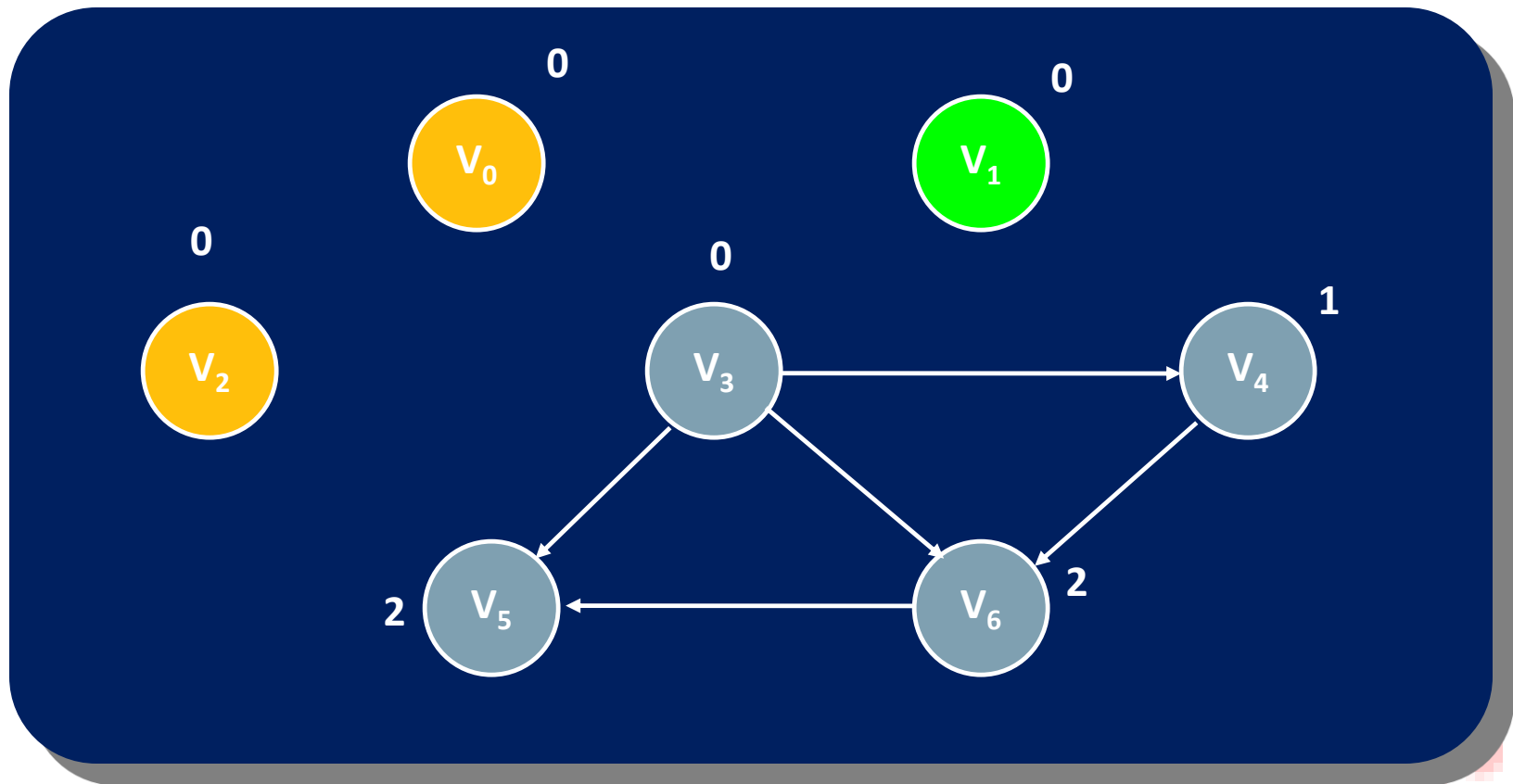
Topological Sorting



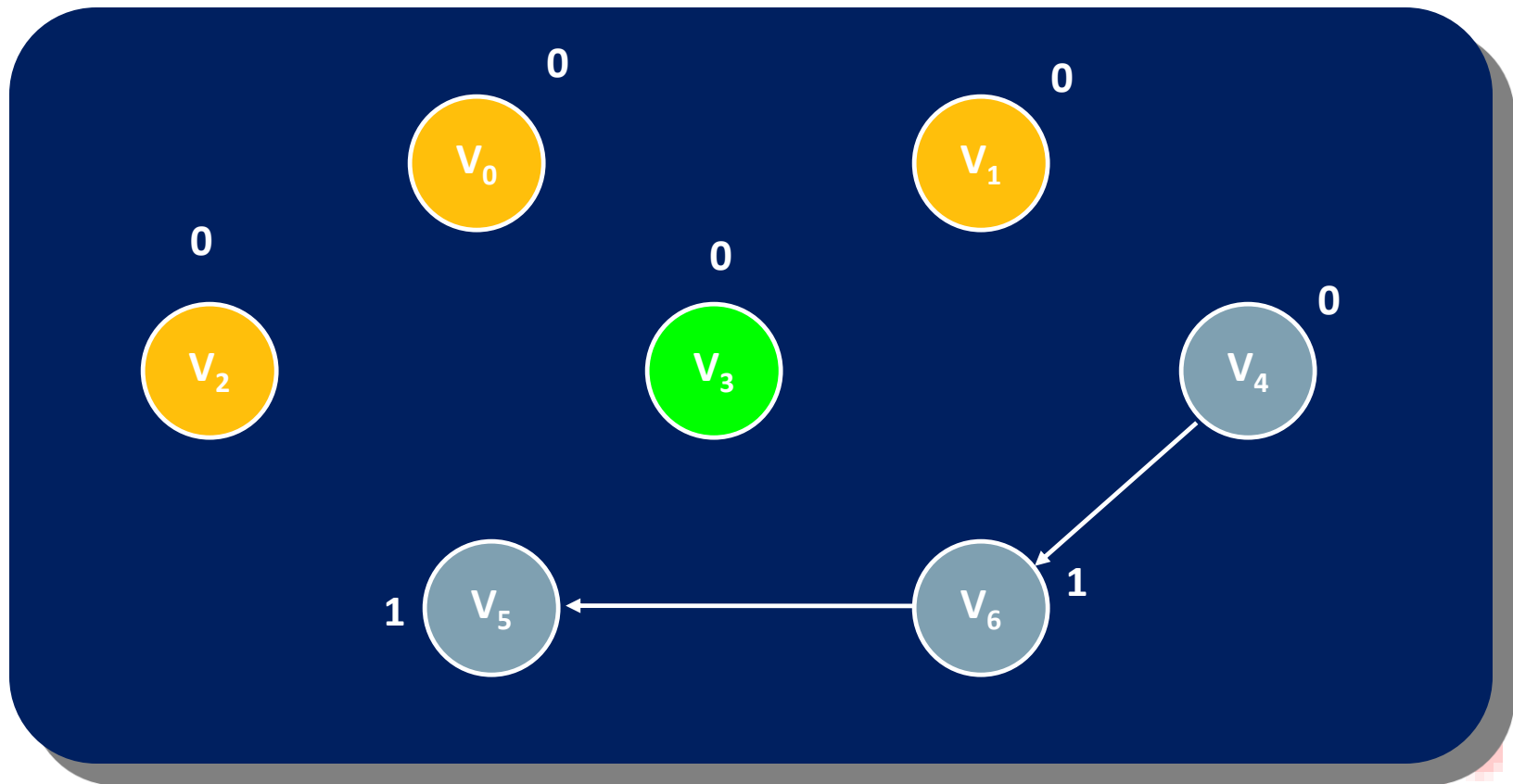
Topological Sorting



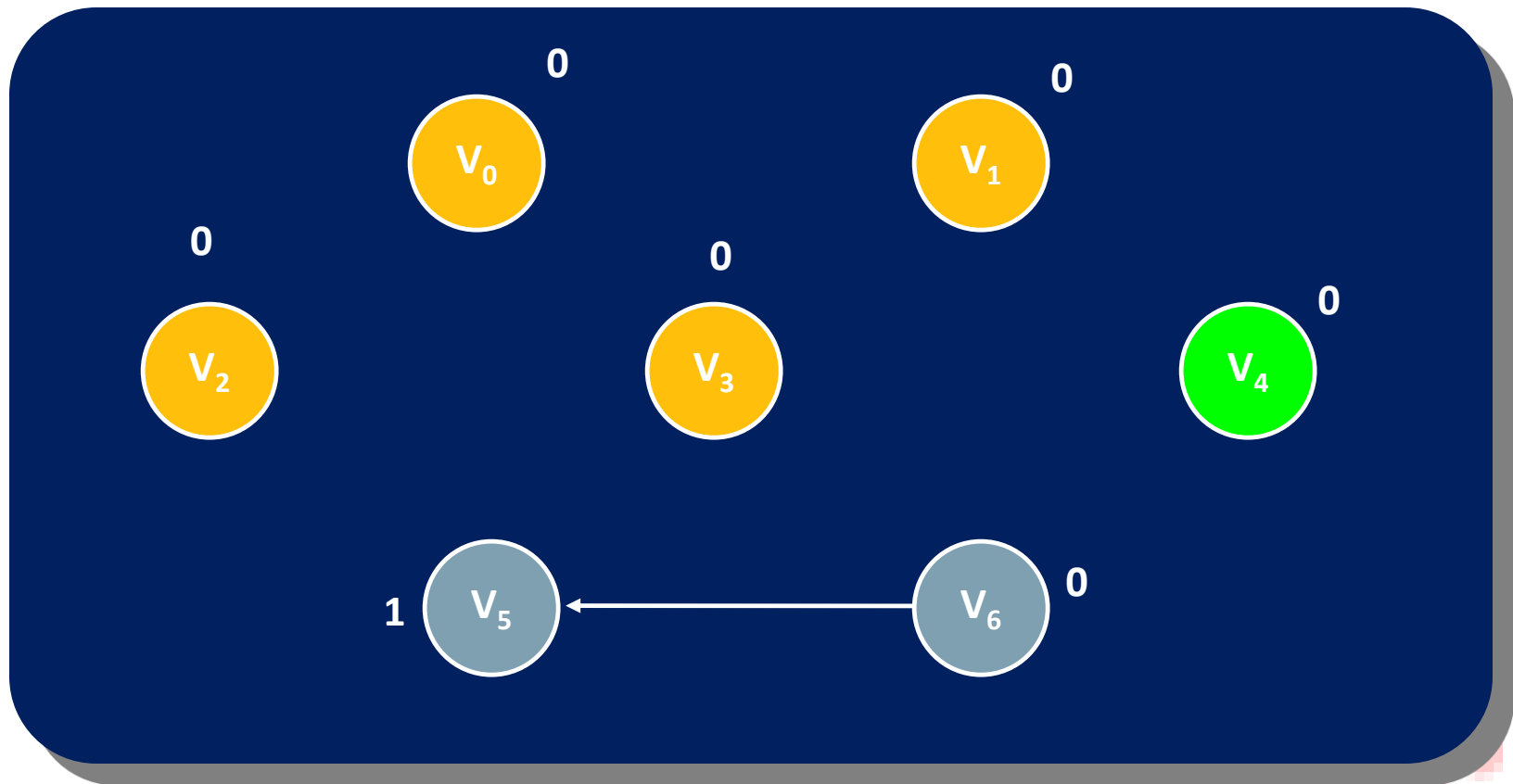
Topological Sorting



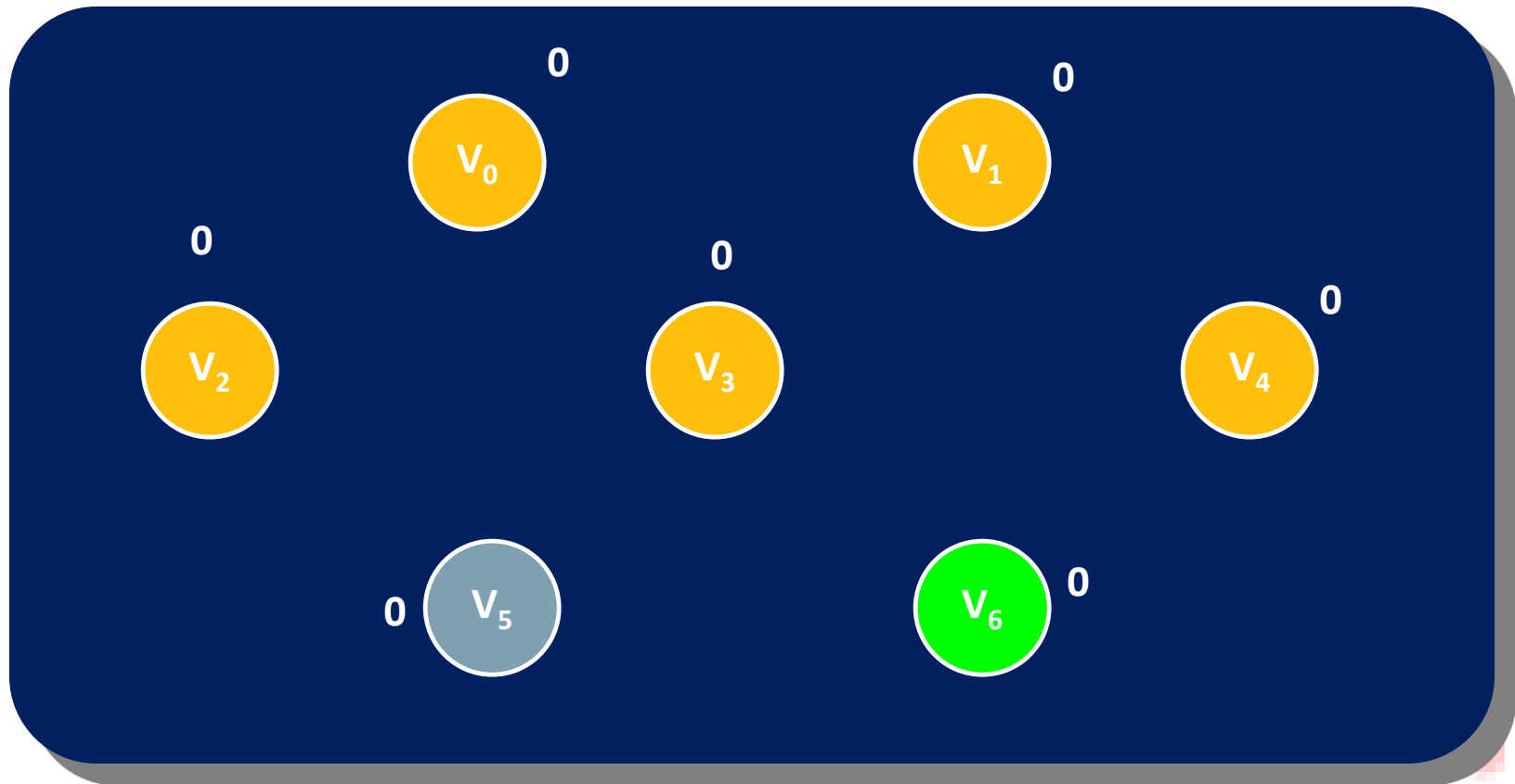
Topological Sorting



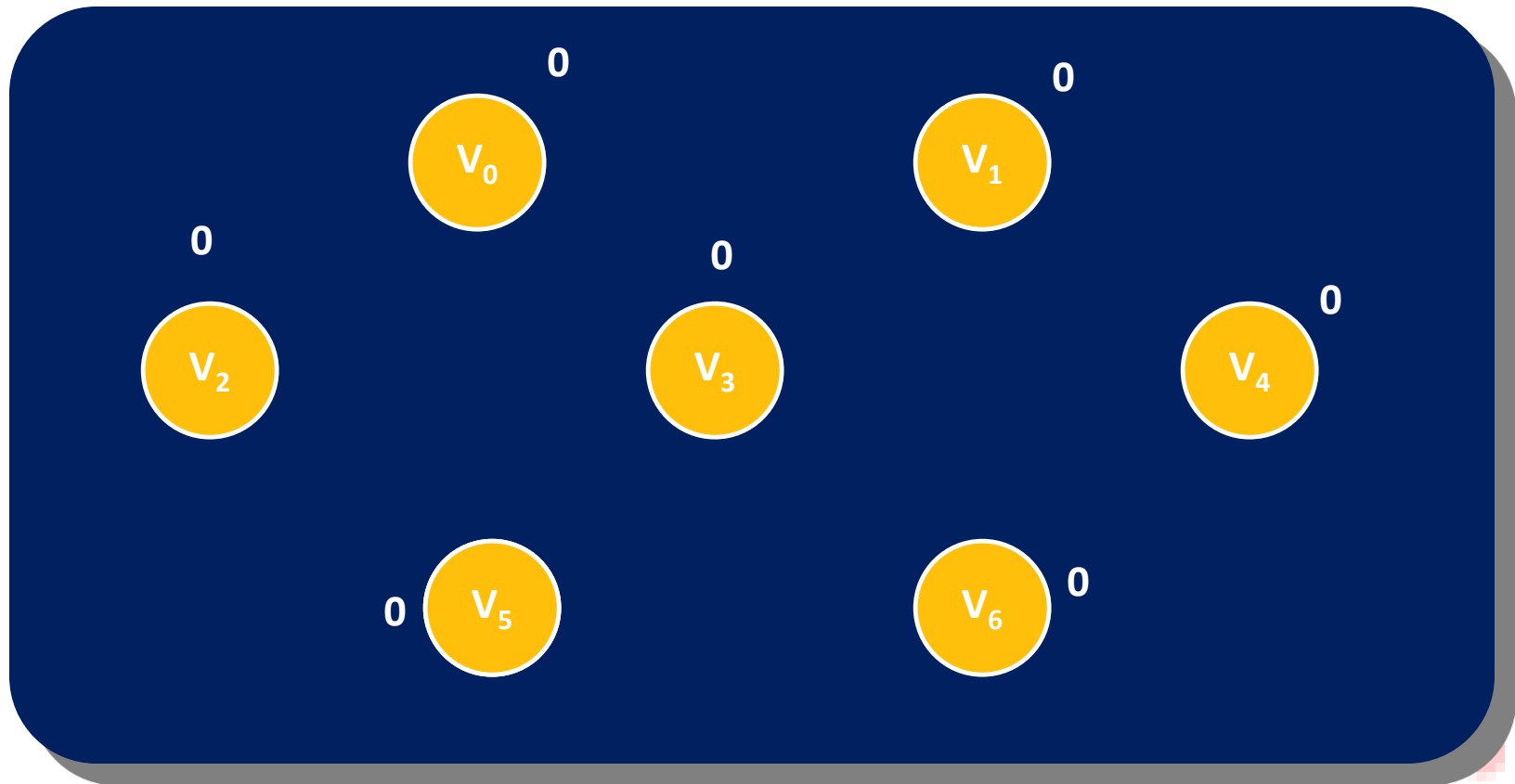
Topological Sorting



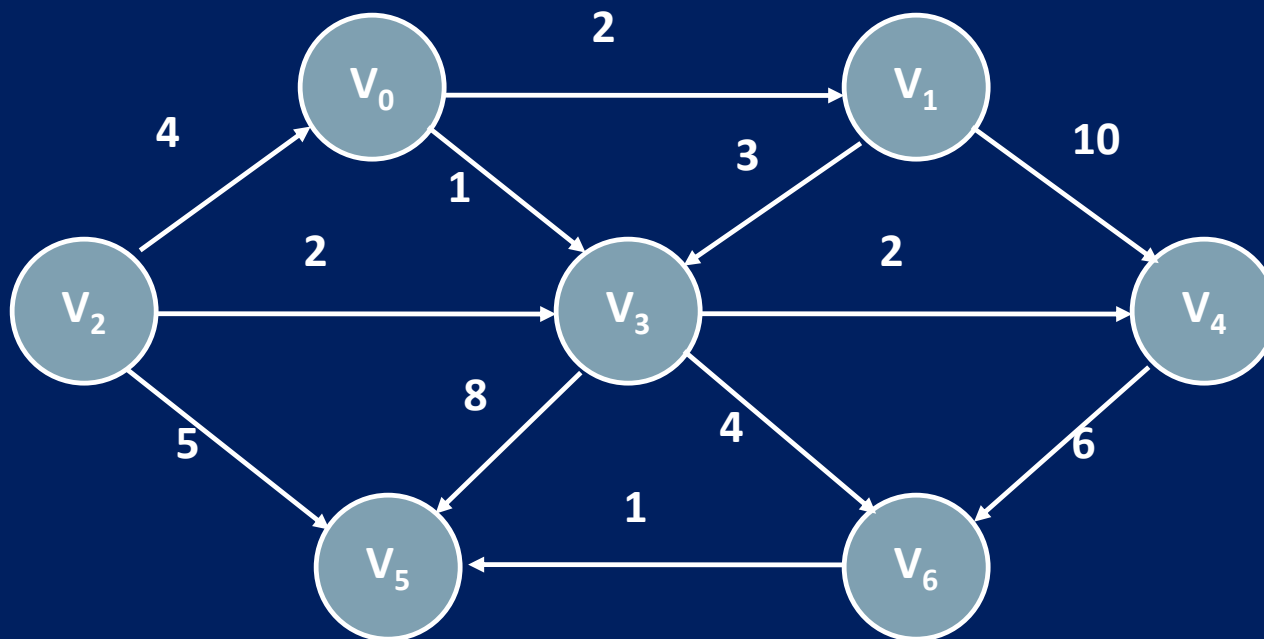
Topological Sorting



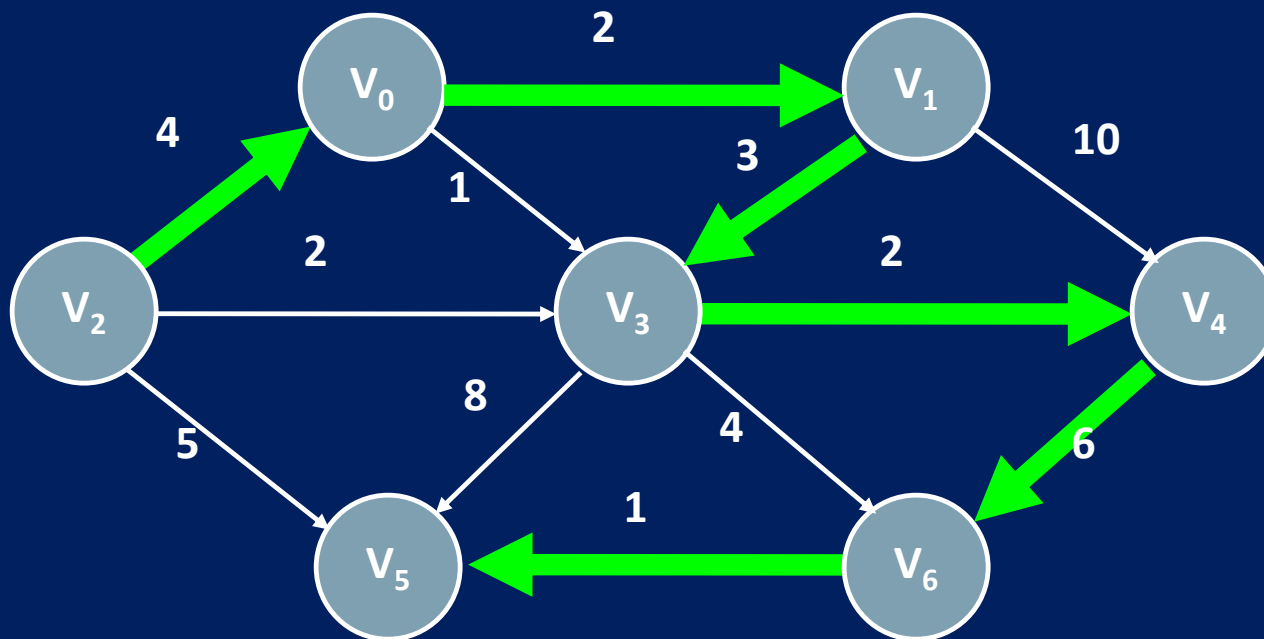
Topological Sorting



Topological Sorting

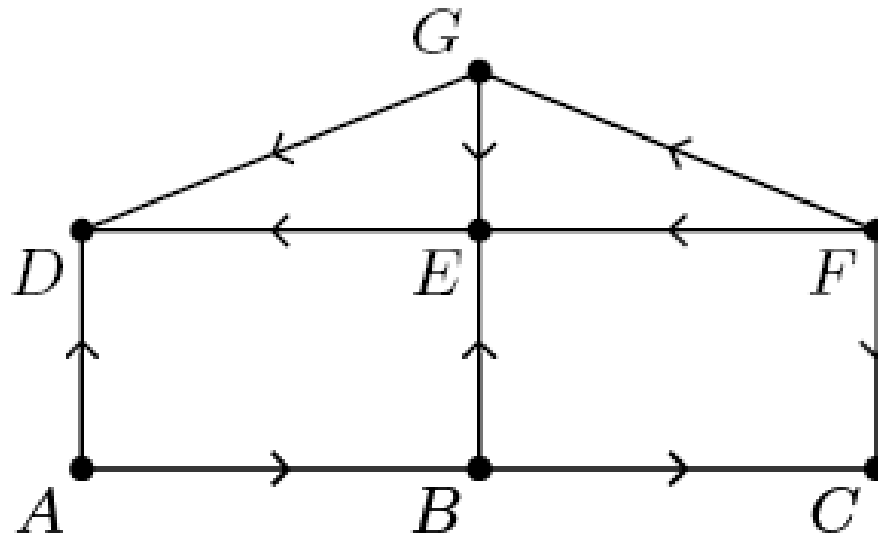


Topological Sorting



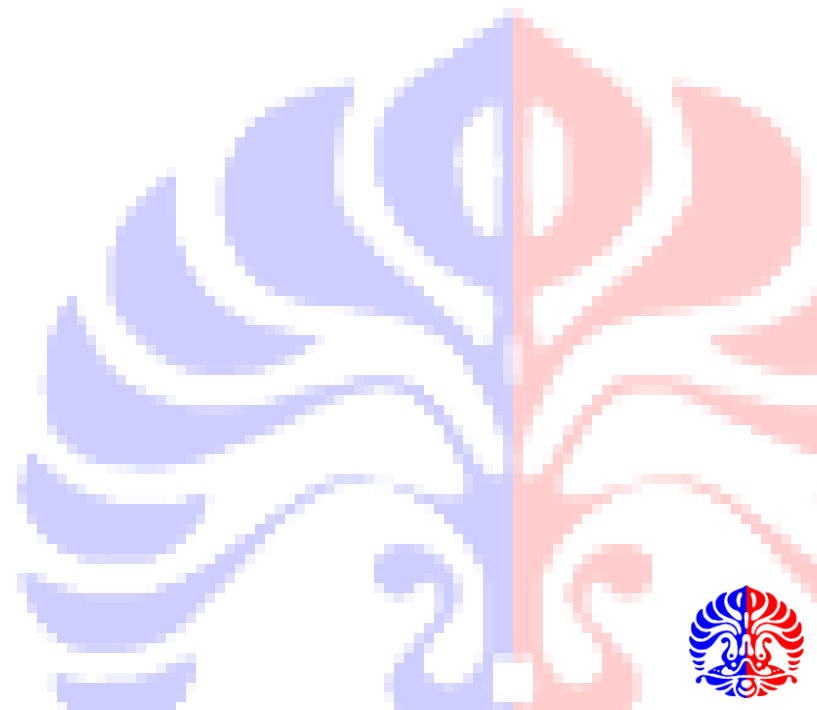
Latihan

- Lakukan topological sorting pada graph berikut ini, apabila ada pilihan node yang memiliki *in-degree* 0, pilih berdasarkan urutan alfabet.



Jawaban

- Urutan pengerjaan:
 - A – B – F – C – G – E – D

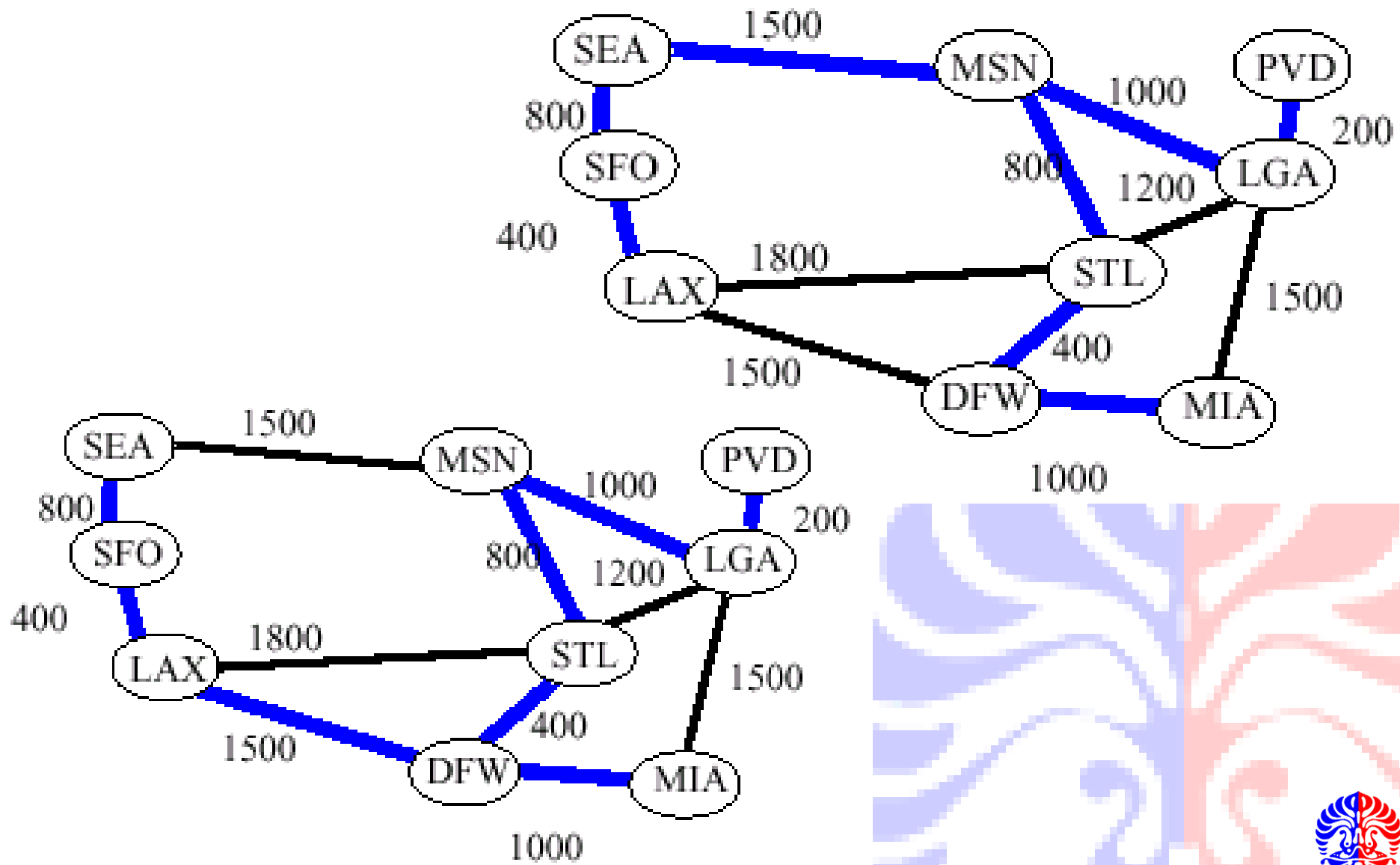


Minimum Spanning Tree (MST)

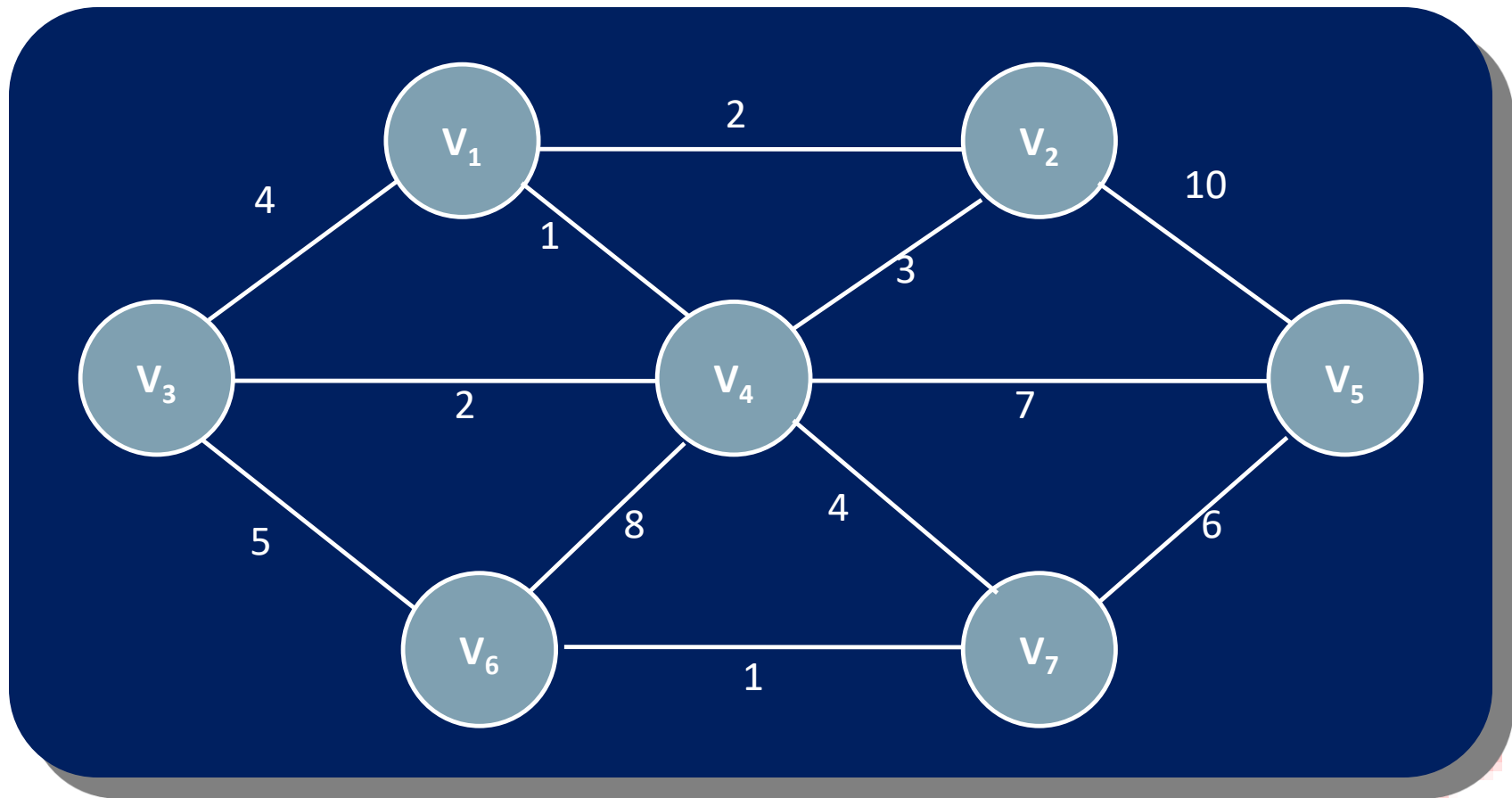
- Adalah sebuah struktur *tree* yang terbentuk dari graph, dimana sisi-sisi yang menghubungkan setiap simpul memiliki nilai total paling kecil.
- “Spanning tree” $T = (V, F)$ dari graph G adalah graph dengan verteks yang sama dengan G dan memiliki $|V|-1$ buah edges, yang membentuk sebuah tree.
- Nilai total dari sebuah spanning tree, adalah jumlah total bobot tiap sisi dalam *tree* tersebut.
- Penerapan:
 - Mencari jumlah biaya kabel paling minimum untuk menghubungkan sebuah kelompok perumahan atau perkotaan.
 - Mencari biaya minimum terendah untuk menghubungkan jaringan komputer.
 - Mencari biaya produksi total terendah untuk pengerjaan proyek.



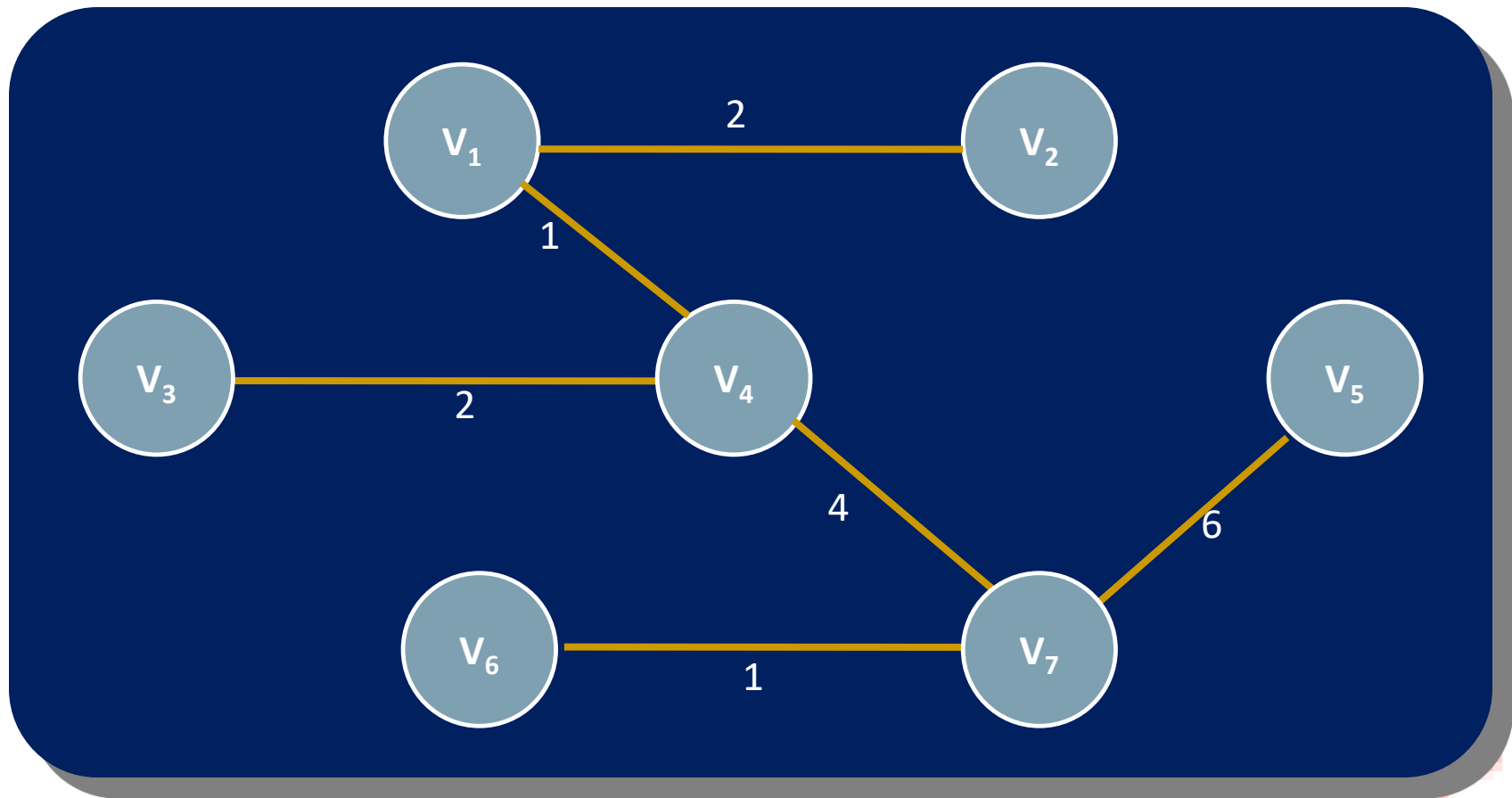
Minimum Spanning Tree (MST)



Minimum Spanning Tree: a graph



Minimum Spanning Tree



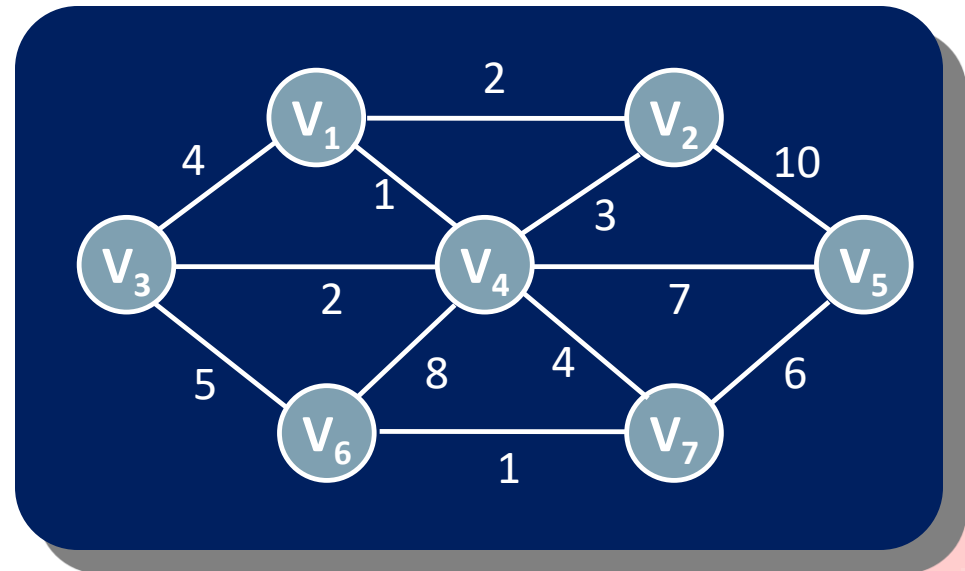
Prim's Algorithm

- mulai dari sebuah simpul
- bangun tree dengan menambahkan sebuah sisi/busur satu persatu.
 - secara berulang pilih sisi terkecil yang dapat menyambung tree.
- greedy algorithms:
 - Pilihan langkah diambil berdasarkan pilihan terbaik secara local tanpa memperhatikan pengaruhnya secara global.



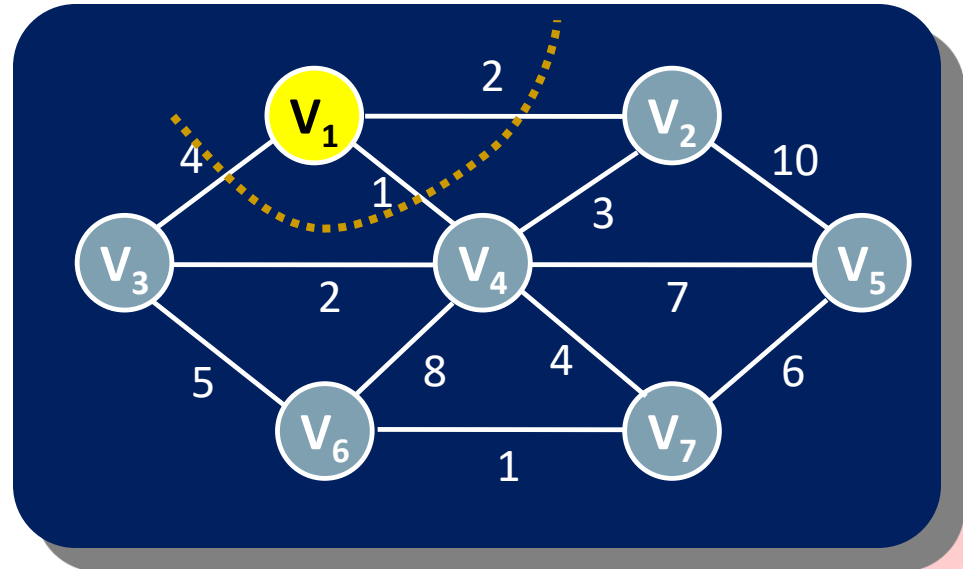
Prim's Algorithm

V	known	d_V	p_V
V₁	0	0	0
V₂	0	∞	0
V₃	0	∞	0
V₄	0	∞	0
V₅	0	∞	0
V₆	0	∞	0
V₇	0	∞	0



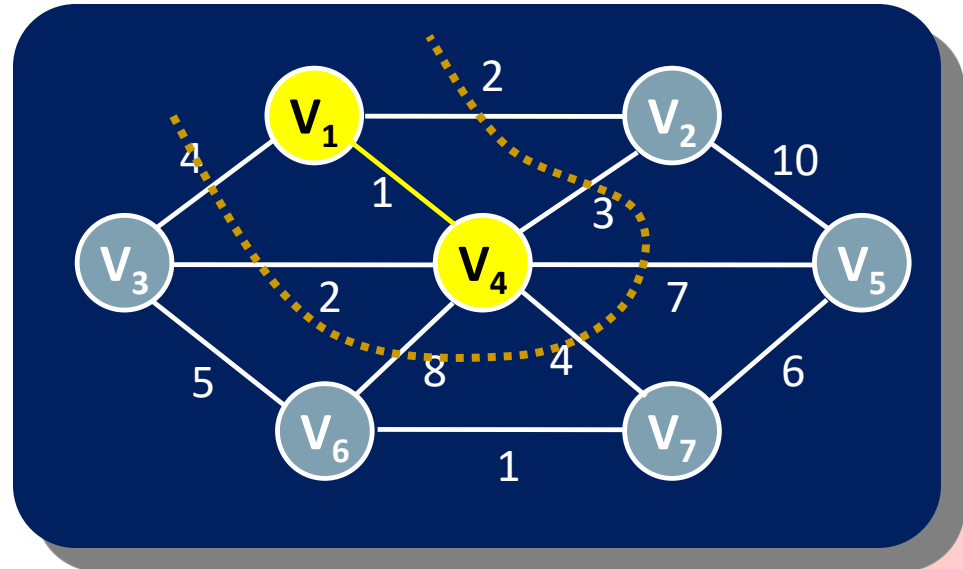
Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	4	V_1
V_4	0	1	V_1
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0



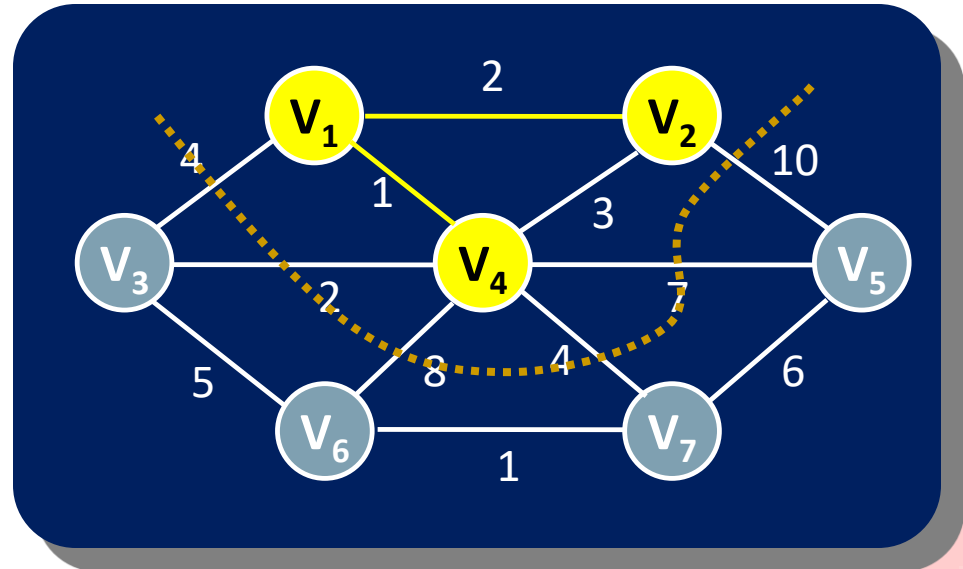
Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	2	V_4
V_4	1	1	V_1
V_5	0	7	V_4
V_6	0	8	V_4
V_7	0	4	V_4



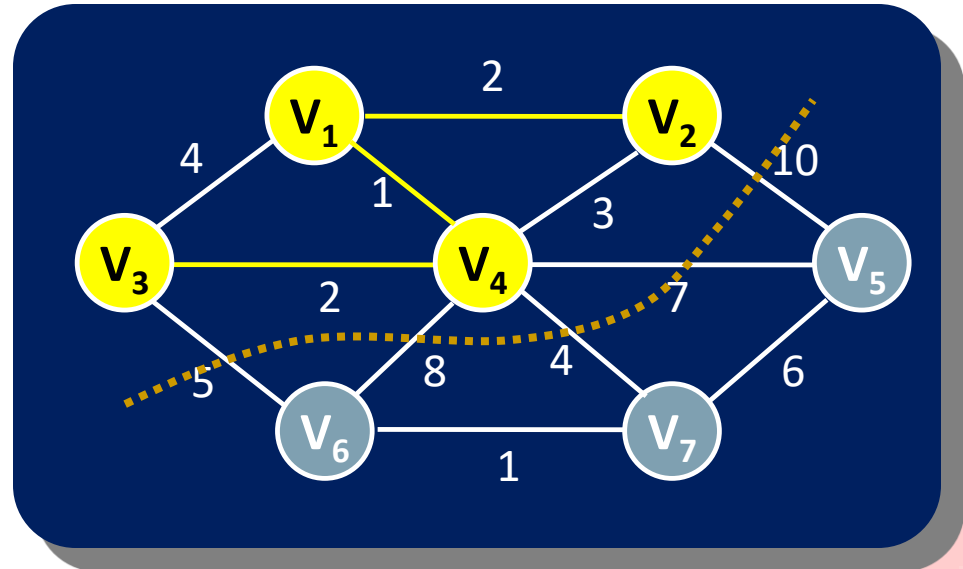
Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	0	2	V_4
V_4	1	1	V_1
V_5	0	7	V_4
V_6	0	8	V_4
V_7	0	4	V_4



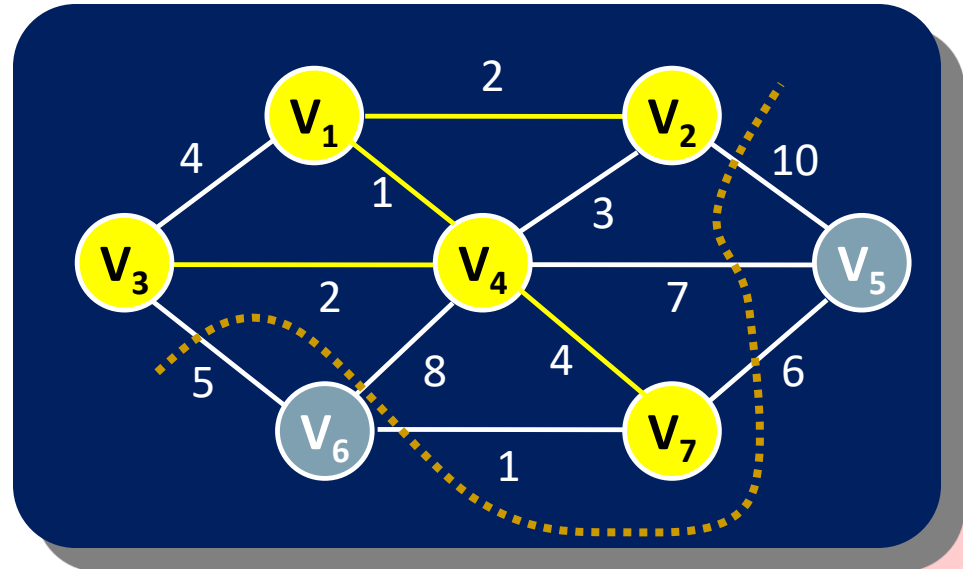
Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	0	7	V_4
V_6	0	5	V_3
V_7	0	4	V_4



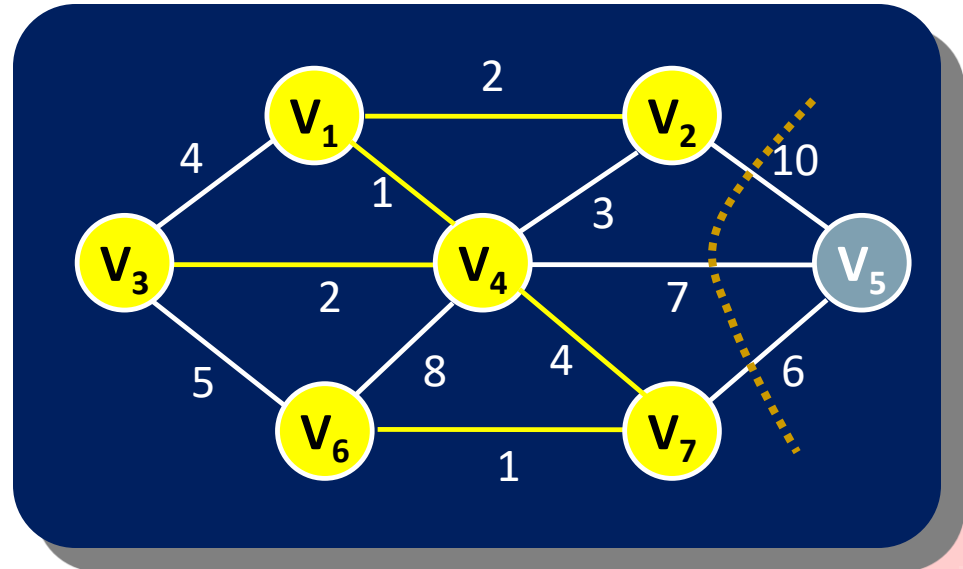
Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	0	6	V_7
V_6	0	1	V_7
V_7	1	4	V_4



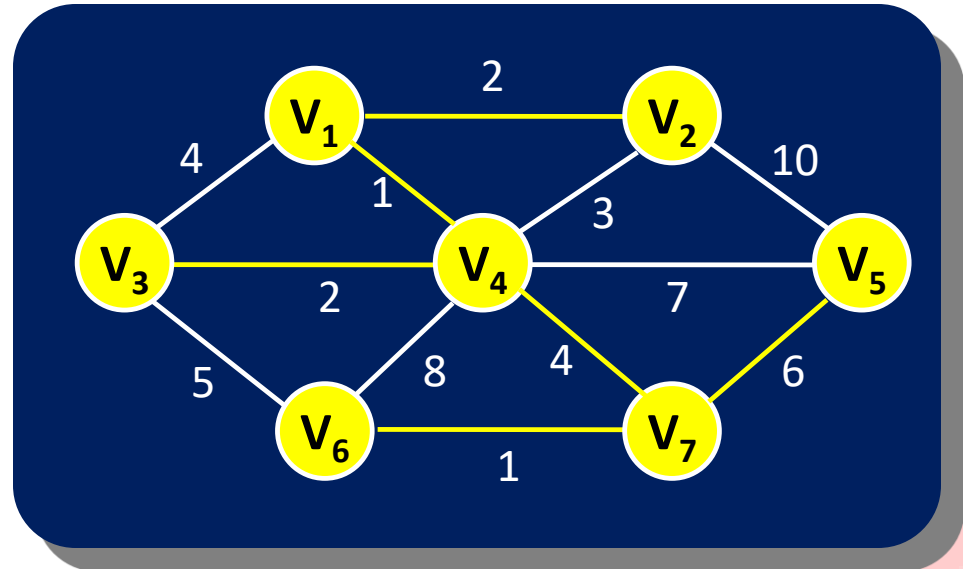
Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	0	6	V_7
V_6	1	1	V_7
V_7	1	4	V_4



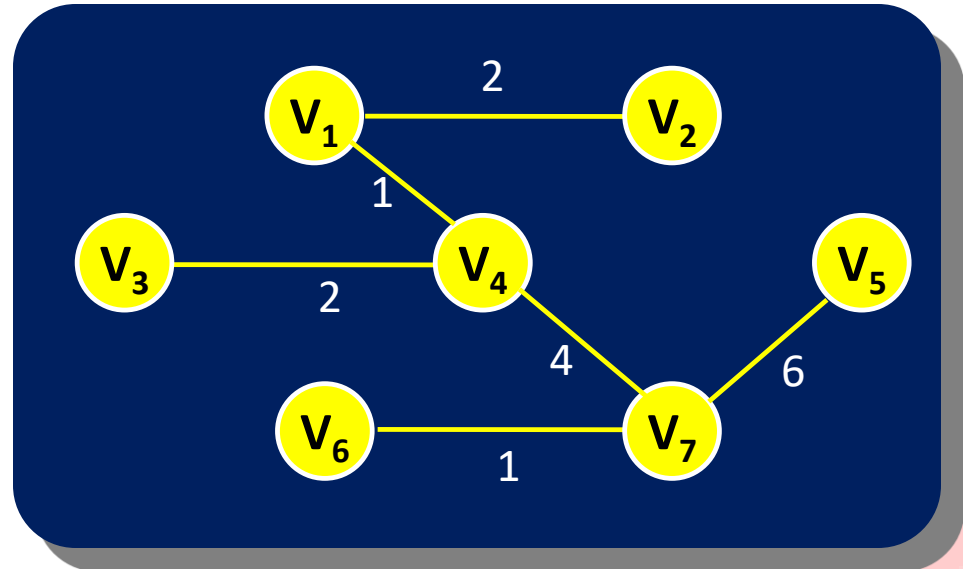
Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	1	6	V_7
V_6	1	1	V_7
V_7	1	4	V_4



Prim's Algorithm

V	known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	1	6	V_7
V_6	1	1	V_7
V_7	1	4	V_4



Kruskal's Algorithm

- Dari sebuah graph $G = (V, E)$, buatlah graph baru T dengan verteks yang sama dengan G namun belum memiliki edges.
- List semua *edges* yang terdapat pada G , urutkan berdasarkan bobot, dari yang terkecil hingga yang terbesar
- Lakukan iterasi untuk setiap *edge* secara terurut. Untuk setiap *edge* (v, u) :
 - Jika u dan v tidak terhubung oleh suatu *path* pada T , tambahkan (u, v) ke dalam T , atau dengan kata lain tambahkan edge ke dalam graph T **apabila tidak menimbulkan cycle.**

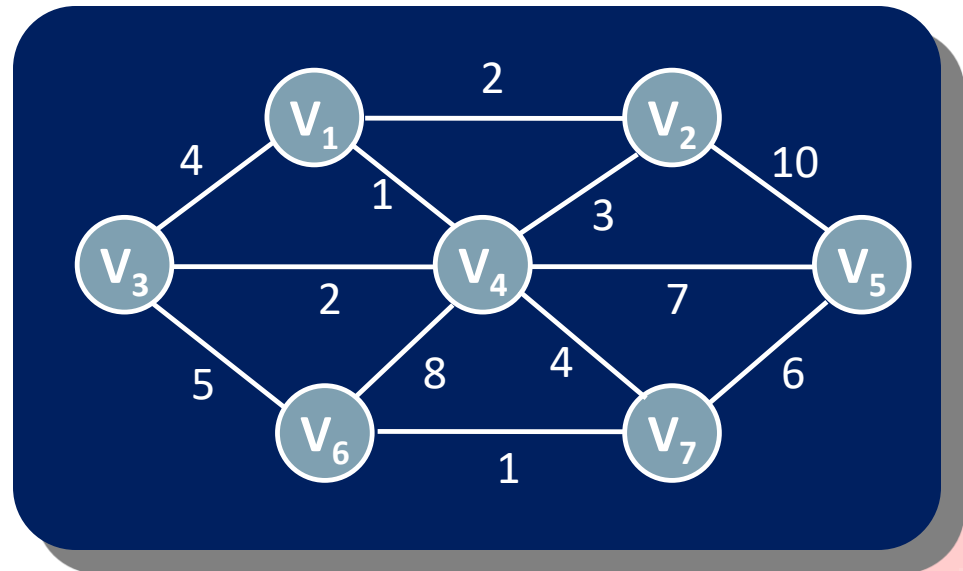
Iterasi dilakukan hingga semua verteks terhubung (jumlah edge = jumlah verteks – 1)

- Pemeriksaan cycle dapat menggunakan struktur data “Union-Find Disjoint Sets”
- Graph T yang terbentuk merupakan MST dari graph G .



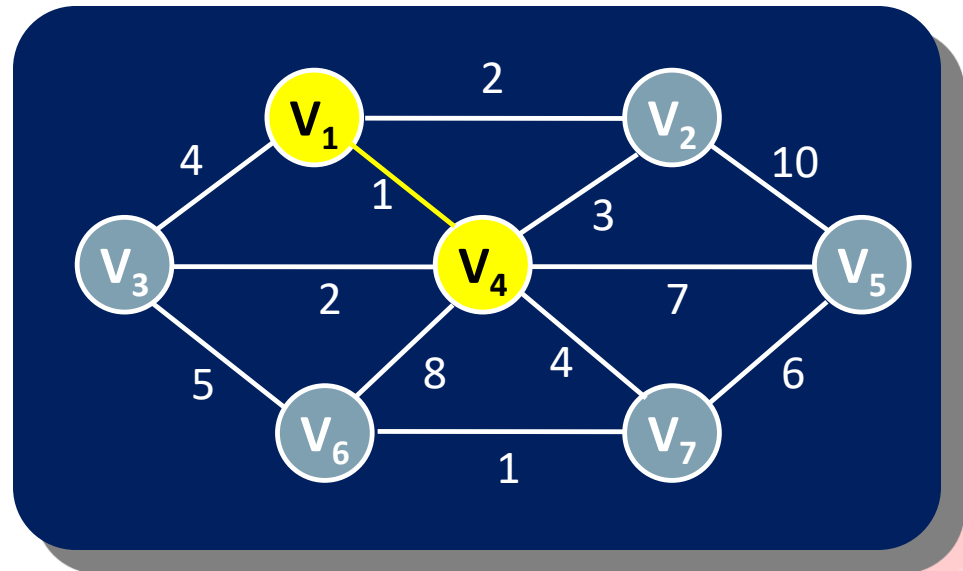
Kruskal's Algorithm

Edge	Weight	Action
(V_1, V_4)	1	-
(V_6, V_7)	1	-
(V_1, V_2)	2	-
(V_3, V_4)	2	-
(V_2, V_4)	3	-
(V_1, V_3)	4	-
(V_4, V_7)	4	-
(V_3, V_6)	5	-
(V_5, V_7)	6	-



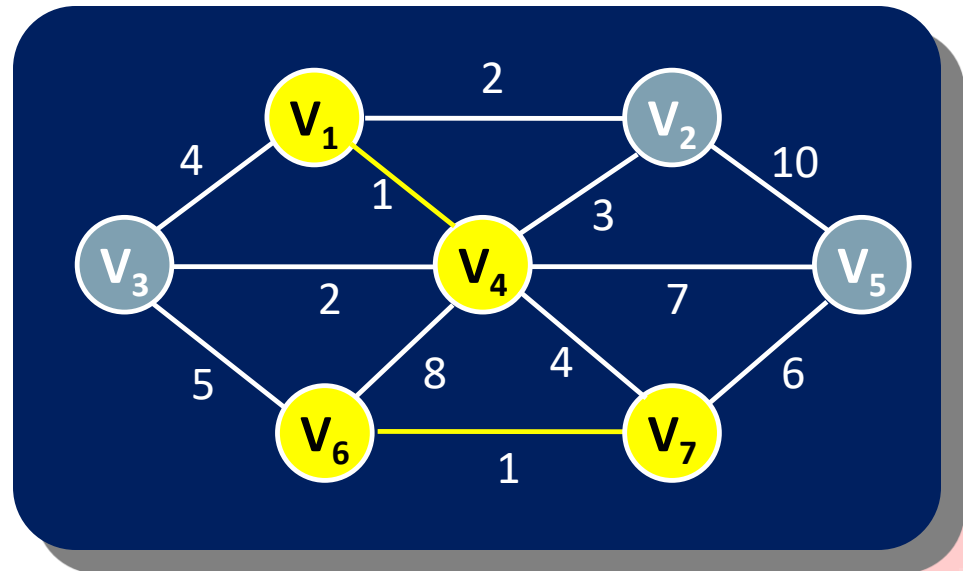
Kruskal's Algorithm

Edge	Weight	Action
(V_1, V_4)	1	A
(V_6, V_7)	1	-
(V_1, V_2)	2	-
(V_3, V_4)	2	-
(V_2, V_4)	3	-
(V_1, V_3)	4	-
(V_4, V_7)	4	-
(V_3, V_6)	5	-
(V_5, V_7)	6	-



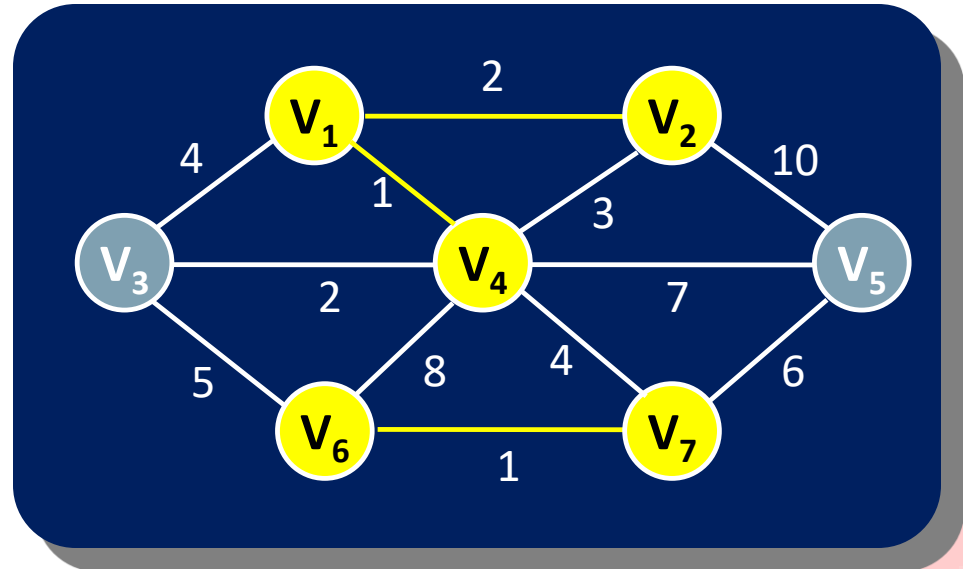
Kruskal's Algorithm

Edge	Weight	Action
(V_1, V_4)	1	A
(V_6, V_7)	1	A
(V_1, V_2)	2	-
(V_3, V_4)	2	-
(V_2, V_4)	3	-
(V_1, V_3)	4	-
(V_4, V_7)	4	-
(V_3, V_6)	5	-
(V_5, V_7)	6	-



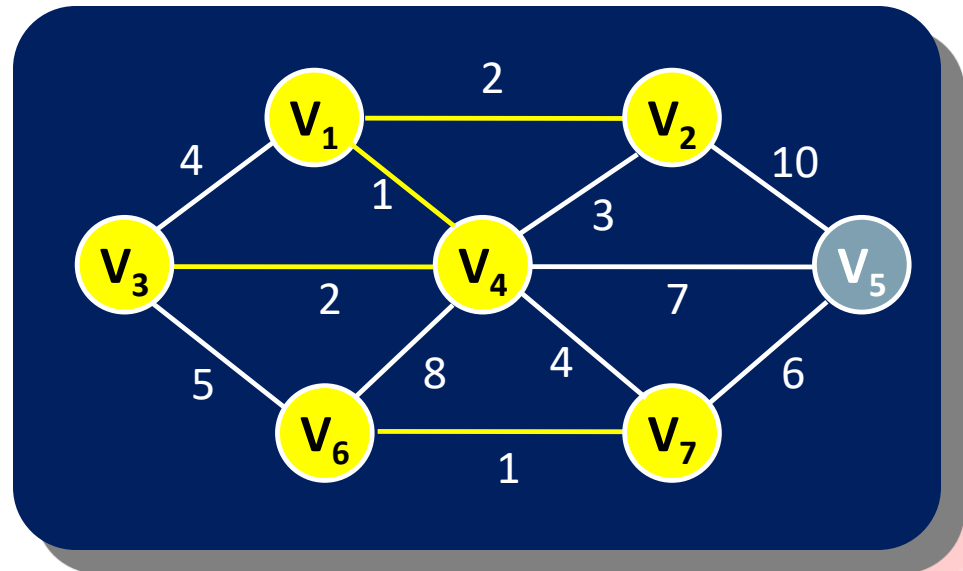
Kruskal's Algorithm

Edge	Weight	Action
(V_1, V_4)	1	A
(V_6, V_7)	1	A
(V_1, V_2)	2	A
(V_3, V_4)	2	-
(V_2, V_4)	3	-
(V_1, V_3)	4	-
(V_4, V_7)	4	-
(V_3, V_6)	5	-
(V_5, V_7)	6	-



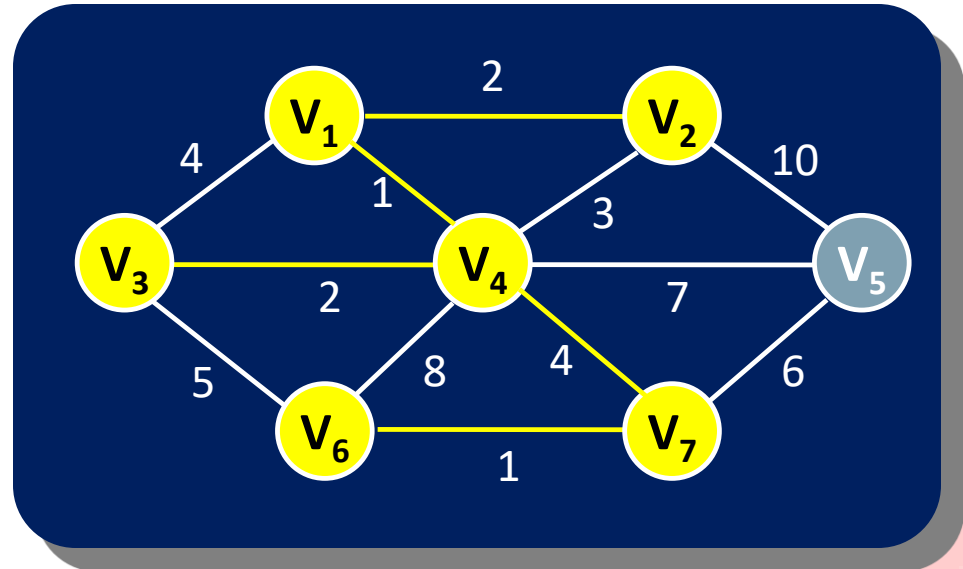
Kruskal's Algorithm

Edge	Weight	Action
(V_1, V_4)	1	A
(V_6, V_7)	1	A
(V_1, V_2)	2	A
(V_3, V_4)	2	A
(V_2, V_4)	3	-
(V_1, V_3)	4	-
(V_4, V_7)	4	-
(V_3, V_6)	5	-
(V_5, V_7)	6	-



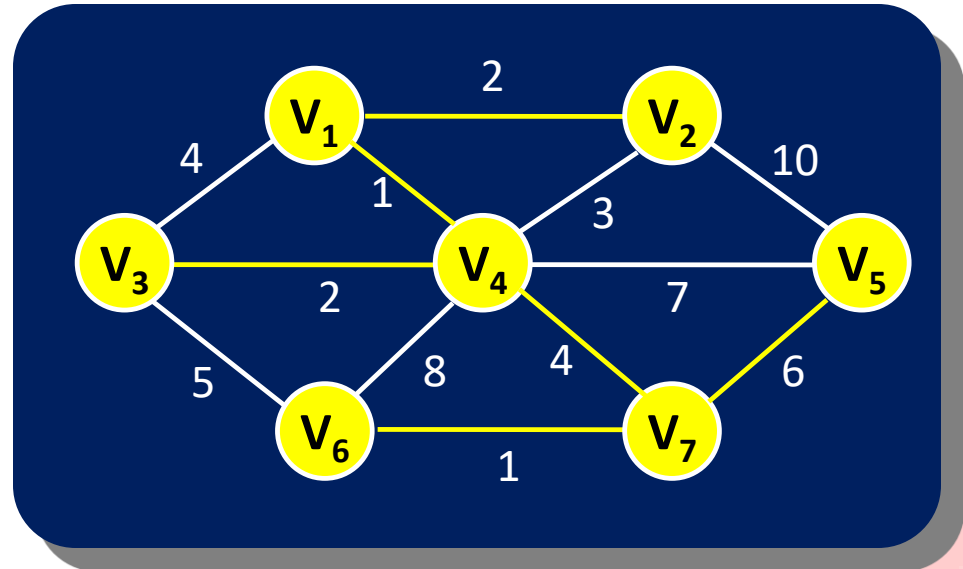
Kruskal's Algorithm

Edge	Weight	Action
(V_1, V_4)	1	A
(V_6, V_7)	1	A
(V_1, V_2)	2	A
(V_3, V_4)	2	A
(V_2, V_4)	3	R
(V_1, V_3)	4	R
(V_4, V_7)	4	A
(V_3, V_6)	5	-
(V_5, V_7)	6	-



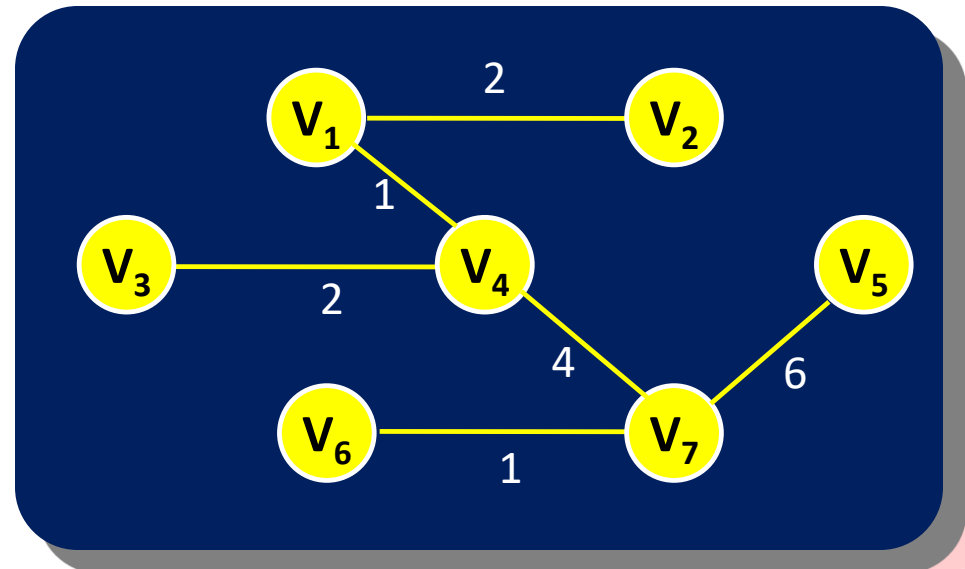
Kruskal's Algorithm

Edge	Weight	Action
(V1, V4)	1	A
(V6, V7)	1	A
(V1, V2)	2	A
(V3, V4)	2	A
(V2, V4)	3	R
(V1, V3)	4	R
(V4, V7)	4	A
(V3, V6)	5	R
(V5, V7)	6	A



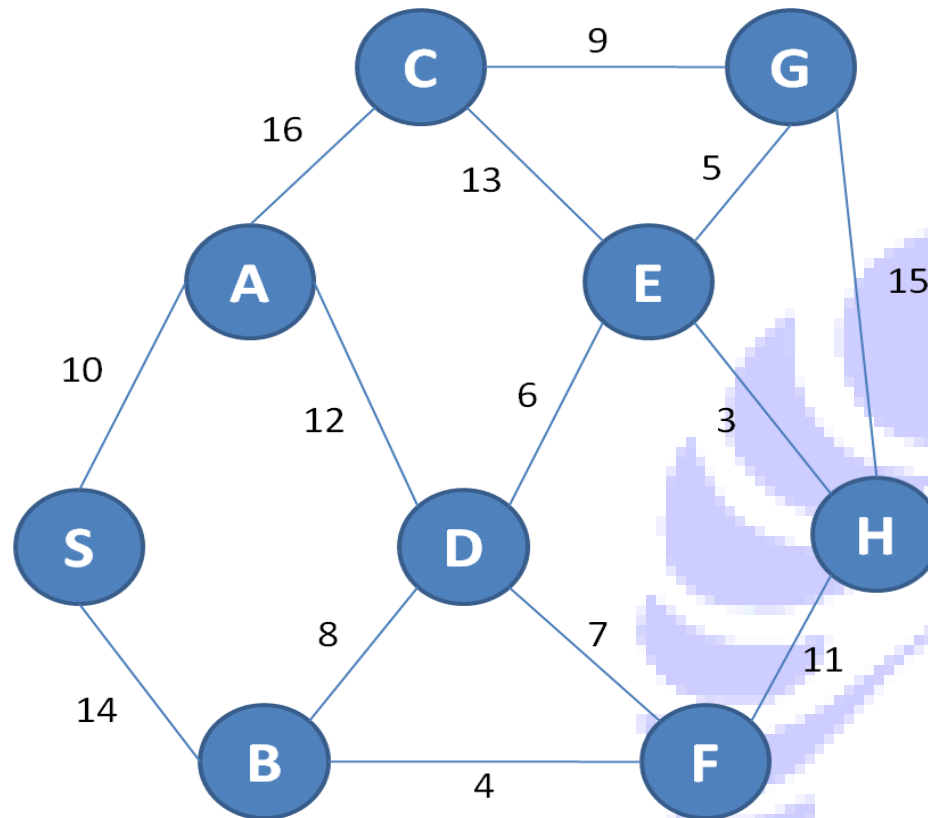
Kruskal's Algorithm

Edge	Weight	Action
(V1, V4)	1	A
(V6, V7)	1	A
(V1, V2)	2	A
(V3, V4)	2	A
(V2, V4)	3	R
(V1, V3)	4	R
(V4, V7)	4	A
(V3, V6)	5	R
(V5, V7)	6	A

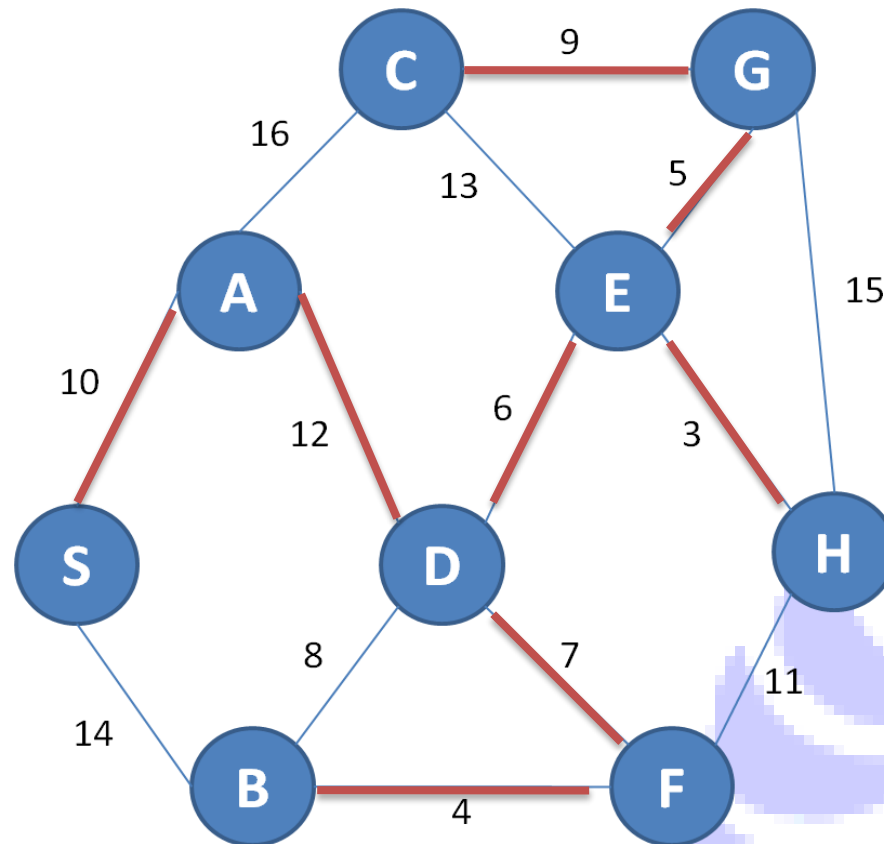


Latihan

- Gambarkan *minimum spanning tree* yang terbentuk dari graph berikut beserta total bobot minimum yang dicapai. Gunakan Prim's dan Kruskal algorithm!



Jawaban



Implementasi Graph

```
1 // Represents an edge in the graph.
2 class Edge
3 {
4     public Vertex dest;      // Second vertex in Edge
5     public double cost;      // Edge cost
6
7     public Edge( Vertex d, double c )
8     {
9         dest = d;
10        cost = c;
11    }
12 }
```

figure 14.6

The basic item stored
in an adjacency list



Implementasi Graph

```
1 // Represents a vertex in the graph.
2 class Vertex
3 {
4     public String    name;    // Vertex name
5     public List<Edge> adj;    // Adjacent vertices
6     public double    dist;    // Cost
7     public Vertex    prev;    // Previous vertex on shortest path
8     public int        scratch; // Extra variable used in algorithm
9
10    public Vertex( String nm )
11        { name = nm; adj = new LinkedList<Edge>( ); reset( ); }
12
13    public void reset( )
14        { dist = Graph.INFINITY; prev = null; pos = null; scratch = 0; }
15 }
```

figure 14.7

The Vertex class stores information for each vertex



Implementasi Graph

figure 14.12

A recursive routine for printing the shortest path

```
1  /**
2   * Recursive routine to print shortest path to dest
3   * after running shortest path algorithm. The path
4   * is known to exist.
5   */
6  private void printPath( Vertex dest )
7  {
8      if( dest.prev != null )
9      {
10         printPath( dest.prev );
11         System.out.print( " to " );
12     }
13     System.out.print( dest.name );
14 }
```



Implementasi Graph

```
1 // Represents an entry in the priority queue for Dijkstra's algorithm.
2 class Path implements Comparable<Path>
3 {
4     public Vertex    dest;    // w
5     public double    cost;    // d(w)
6
7     public Path( Vertex d, double c )
8     {
9         dest = d;
10        cost = c;
11    }
12
13    public int compareTo( Path rhs )
14    {
15        double otherCost = rhs.cost;
16
17        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
18    }
19 }
```

figure 14.26

Basic item stored in the priority queue


```

1  /**
2   * Single-source weighted shortest-path algorithm.
3   */
4  public void dijkstra( String startName )
5  {
6      PriorityQueue<Path> pq = new PriorityQueue<Path>( );
7
8      Vertex start = vertexMap.get( startName );
9      if( start == null )
10         throw new NoSuchElementException( "Start vertex not found" );
11
12     clearAll( );
13     pq.add( new Path( start, 0 ) ); start.dist = 0;
14
15     int nodesSeen = 0;
16     while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
17     {
18         Path vrec = pq.remove( );
19         Vertex v = vrec.dest;
20         if( v.scratch != 0 ) // already processed v
21             continue;
22
23         v.scratch = 1;
24         nodesSeen++;
25
26         for( Edge e : v.adj )
27         {
28             Vertex w = e.dest;
29             double cvw = e.cost;
30
31             if( cvw < 0 )
32                 throw new GraphException( "Graph has negative edges" );
33
34             if( w.dist > v.dist + cvw )
35             {
36                 w.dist = v.dist + cvw;
37                 w.prev = v;
38                 pq.add( new Path( w, w.dist ) );
39             }
40         }
41     }
42 }

```

figure 14.27

A positive-weighted, shortest-path algorithm: Dijkstra's algorithm



Extra material

UNION FIND DISJOINT SET



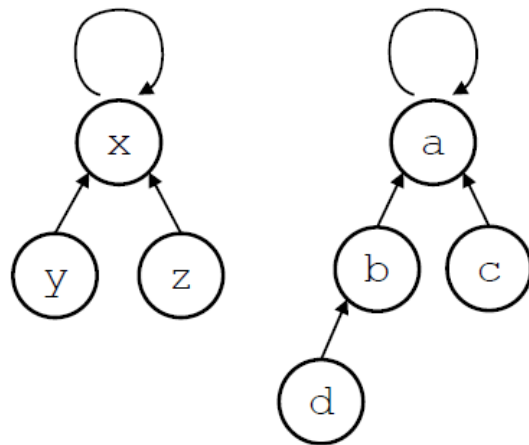
Union-Find Disjoint Sets

- Can support two types of operations efficiently
 - $\text{find}(x)$: returns the “representative” of the set that x belongs
 - $\text{union}(x, y)$: merges two sets that contain x and y
- Both operations can be done in (essentially) constant time
- Simple and short implementation!
- Applications:
 - track connected components of an undirected graph
 - used for implementing Kruskal's algorithm to find the minimum spanning tree of a graph.
 - connecting an edge from two connected vertices will create a cycle



Data Structure

- Main idea: represent each set by a rooted tree
 - Every node maintains a link to its parent
 - initially, each node points to itself
 - A root node is the “representative” of the corresponding set
 - Example: two sets $\{x, y, z\}$ and $\{a, b, c, d\}$



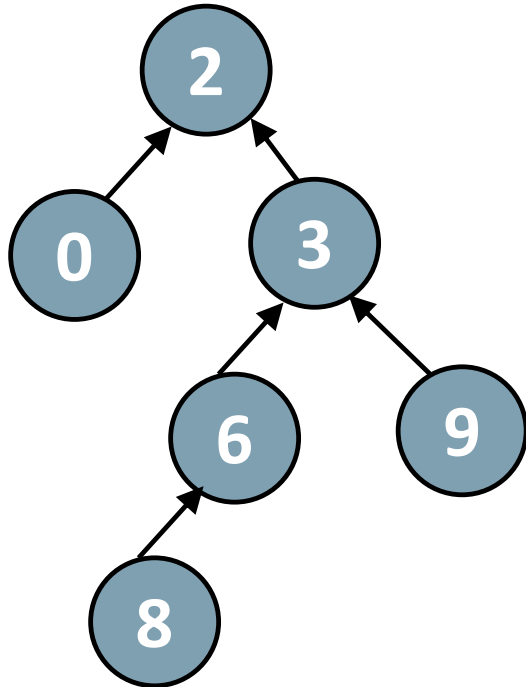
Implementation

- `find(x)`: follow the links from `x` until a node points itself (*the representative node*)
 - This can take $O(n)$ time but we will make it faster
 - Path compression
 - The shape of the tree is not important as long as the root stays the same
 - After `find(x)` returns the root, backtrack to `x` and reroute all the links to the root
- `union(x, y)`: run `find(x)` and `find(y)` to find corresponding root nodes and direct one to the other
 - Another improvement: Union by rank: attach the smaller tree to the larger tree

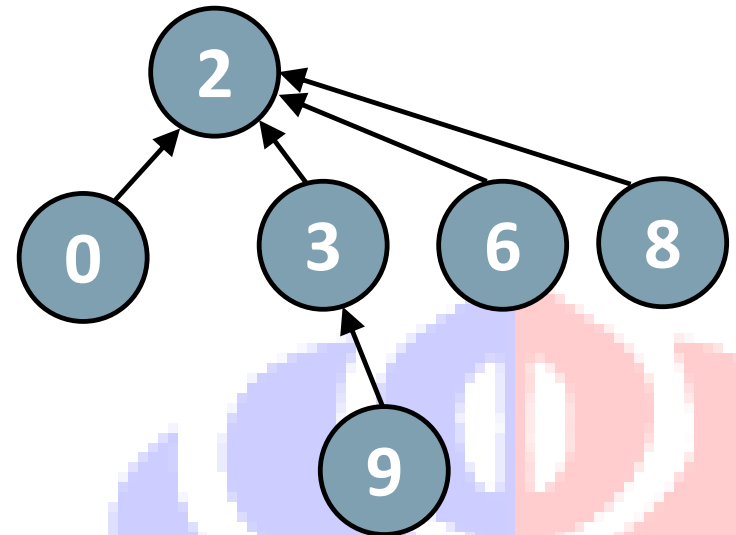


Path Compression

Initially



After calling findSet (8)



Java Implementation

```
public class UnionFindDisjointSet
{
    private int parent[];

    public UnionFindDisjointSet (int size) {
        parent = new int[size];
        for (int ii = 0; ii < size; ii++) {
            parent[ii] = ii;
        }
    }

    public int findSet (int i) {
        if (parent[i] == i) {
            return i;
        } else {
            return parent[i] = findSet (parent[i]); // path compression
        }
    }

    public void unionSet (int i, int j) {
        parent[findSet (i)] = findSet (j);
    }

    public boolean isSameSet (int i, int j) {
        return findSet (i) == findSet (j);
    }
}
```

