Fall 2019

Graphs - Part:1

CMPE 250 - Data Structures & Algorithms

Presenter: Meriç Turan

14 NOVEMBER 2019

Outline

- Graph Definitions
- Topological Sort
- Breadth First Search (BFS)

Definitions

- A graph G = (V, E) consists of a set of **vertices** V, and a set of **edges** E.
- Each edge is a pair (v, w), where v, w ∈ V.
- Edges are sometimes referred to as **arcs**.
- If the pair is ordered, then the graph is **directed**.
- Directed graphs are sometimes referred to as **digraphs**.
- Vertex w is **adjacent** to v if and only if (v, w) ∈ E.
- In an undirected graph with edge (v, w), and hence (w, v), w is adjacent to v and v is adjacent to w.
- Sometimes an edge has a third component, known as either a **weight** or a **cost**.

Definitions

- A **path** in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$.
- The **length** of such a path is the number of edges on the path, which is equal to $N - 1$.
- We allow a path from a vertex to itself; if this path contains no edges, then the path length is 0.
- If the graph contains an edge (v, v) from a vertex to itself, then the path v, v is sometimes referred to as a **loop**.
- The graphs we will consider will generally be loopless.
- A **simple path** is a path such that all vertices are distinct, except that the first and last could be the same.

Definitions

- The path u, v, u in an undirected graph should not be considered a cycle, because (u, v) and (v, u) are the same edge. In a directed graph, these are different edges, so it makes sense to call this a **cycle**.
- A directed graph is **acyclic** if it has no cycles.
- A directed acyclic graph is sometimes referred to by its abbreviation, **DAG**.
- An undirected graph is **connected** if there is <u>a path from every vertex to every other vertex</u>.
- A directed graph with this property is called **strongly connected**.
- If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be **weakly connected**.
- A **complete graph** is a graph in which there is an <u>edge between every pair of vertices</u>.

Example (Airport System)

- Each **airport** is a **vertex**, and **two vertices** are **connected by an edge** if there is a nonstop flight from the airports that are represented by the vertices.
- Edge could have a **weight**, representing the **time**, **distance**, or **cost** of the flight.
- It is reasonable to assume that such a graph is **directed**, since it might **take longer** or **cost more** (e.g., depending on local taxes) to fly in different directions.
- We would probably like to make sure that the airport system is **strongly connected**, so that it is always possible to fly from any airport to any other airport.
- We might also like to quickly determine the **best** flight between any two airports.
- "Best" could mean the path with the fewest number of edges or could be taken with respect to one, or all, of the weight measures.

Example (Traffic Flow)

- Each street intersection represents a vertex, and each street is an edge.
- The edge **cost**s could represent, among other things, a **speed limit** or a **capacity** (number of lanes).
- We could then ask for the **shortest route** or use this information to find the most likely location for bottlenecks.

Sample Directed Graph

The below graph represents 7 vertices and 12 edges.

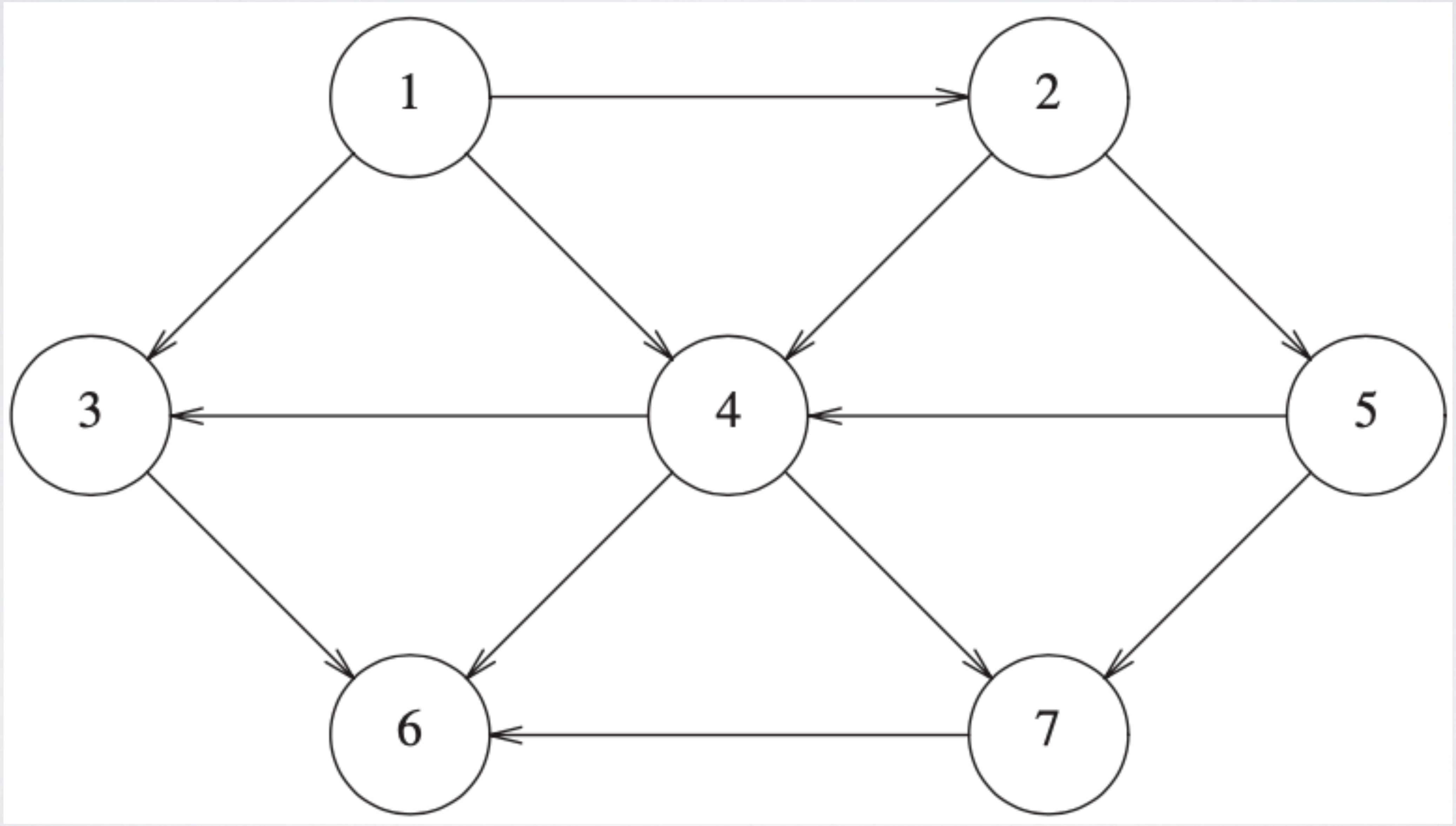
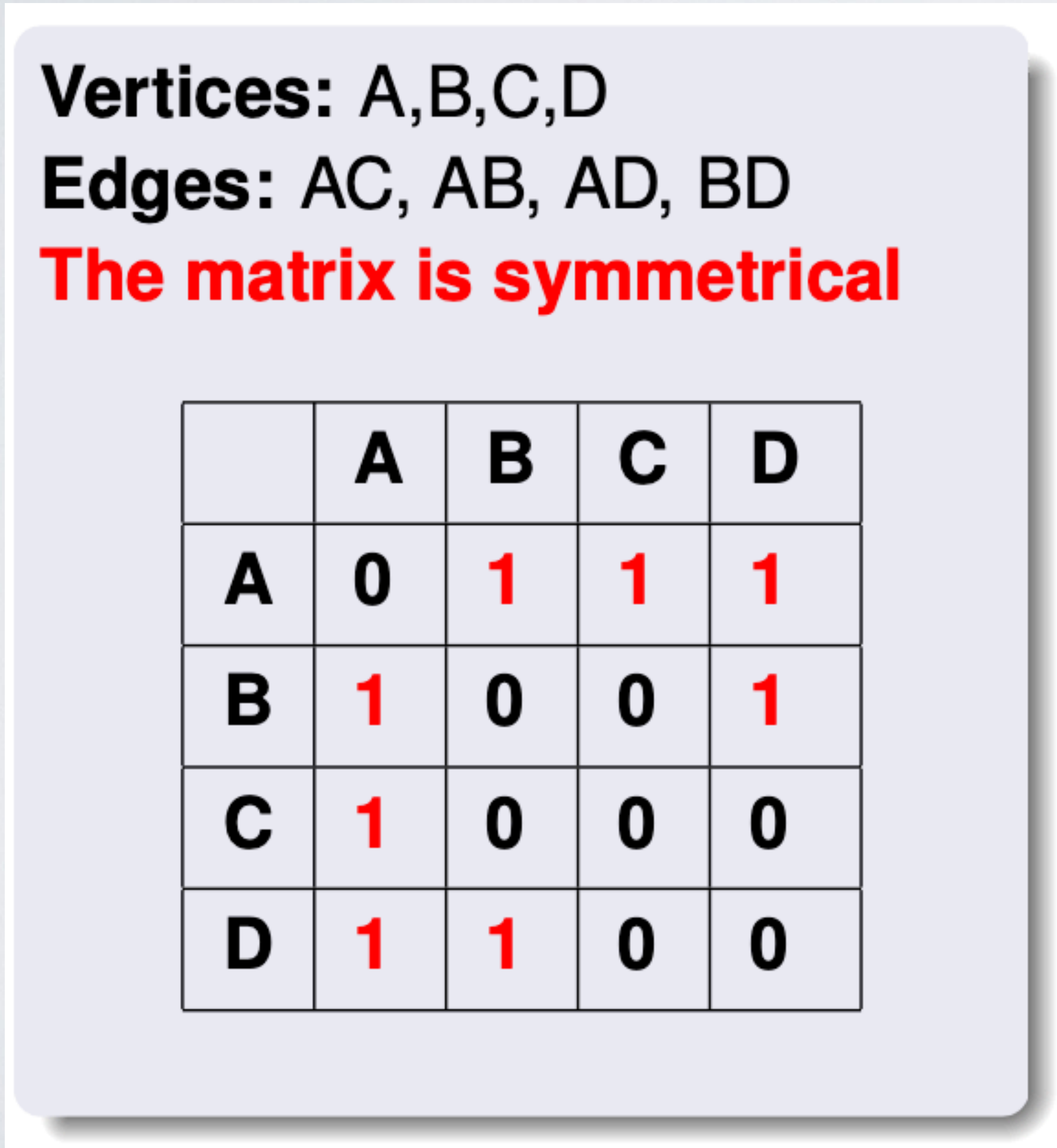


Figure: A directed graph.

Adjacency Matrix

- One simple way to represent a graph is to use a two-dimensional array. This is known as an **adjacency matrix** representation.
- For each edge (u, v), we set A[u][v] to **true**; otherwise the entry in the array is **false**.
- If the edge has a weight associated with it, then we can set A[u][v] equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.
- For instance, if we were looking for the cheapest airplane route, we could represent nonexistent flights with a cost of ∞.
- If we were looking, for some strange reason, for the most expensive airplane route, we could use $-\infty$ (or perhaps 0) to represent nonexistent edges.

Adjacency Matrix for Undirected Graphs

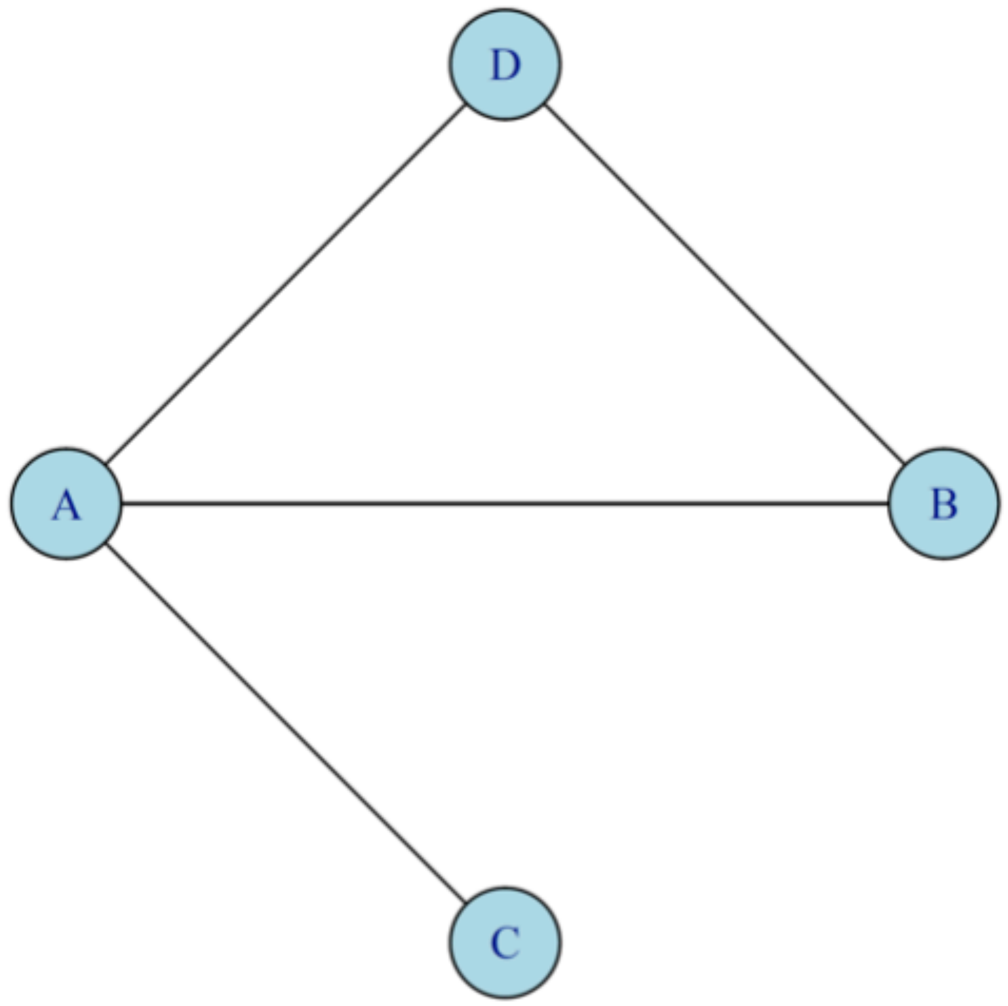


Vertices: A,B,C,D

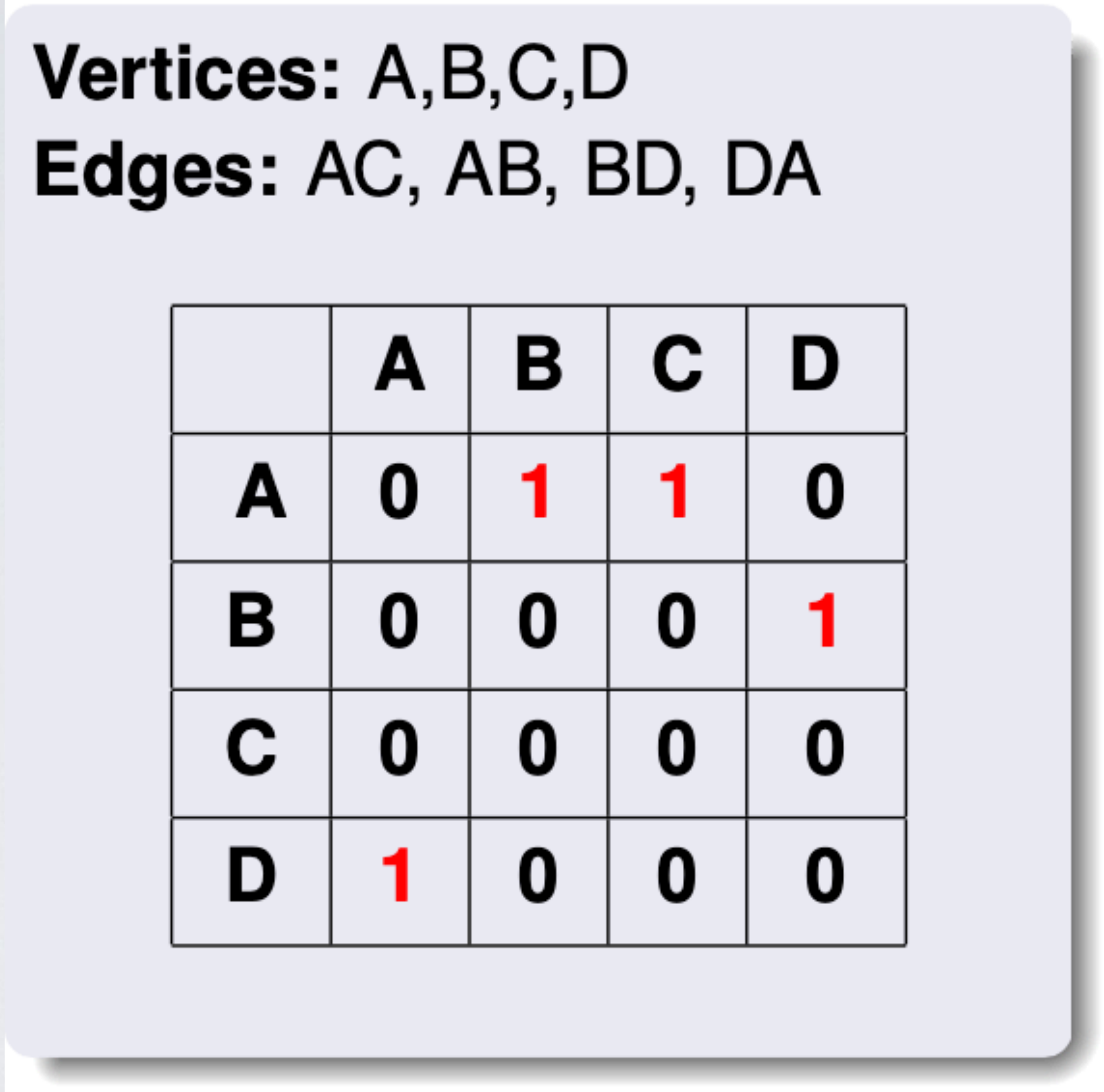
Edges: AC, AB, AD, BD

The matrix is symmetrical

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

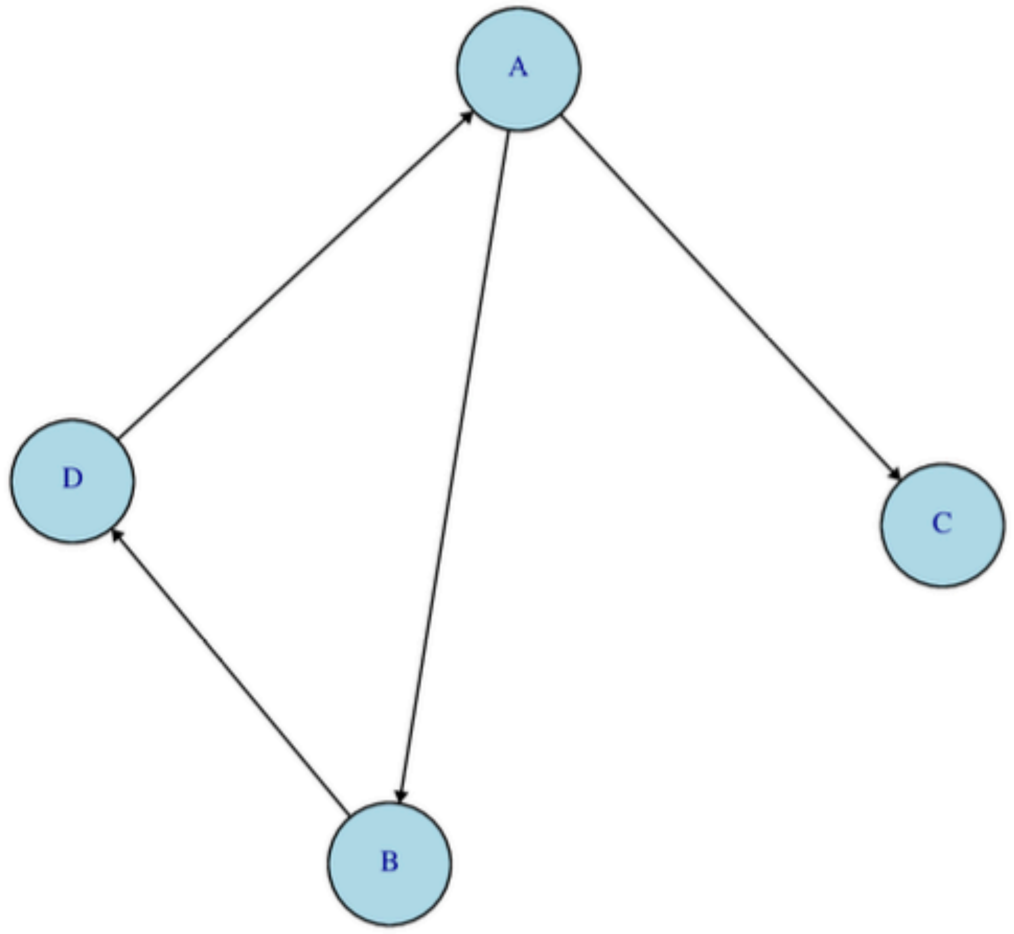


Adjacency Matrix for Directed Graphs



Vertices: A,B,C,D
Edges: AC, AB, BD, DA

	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	1	0	0	0



Adjacency Matrix

- Although this has the merit of extreme simplicity, the space requirement is $O(|V|^2)$, which can be prohibitive if the graph does not have very many edges.
- An adjacency matrix is an appropriate representation if the graph is **dense**: $|E| = O(|V|^2)$.
- In most of the applications that we shall see, this is not true.
- For instance, suppose the graph represents a street map. Assume a Manhattan-like orientation, where almost all streets run either north-south or east–west. Therefore, any intersection is attached to roughly four streets, so if graph is directed and all streets are two-way, then $|E| \approx 4|V|$.
- If there are 3,000 intersections, then we have a 3,000-vertex graph with 12,000 edge entries, which would require an array of size 9,000,000. Most of these entries would contain zero. This is intuitively bad, because we want our data structures to represent the data that are actually there and not the data that are not present.

Graphs - Part:1

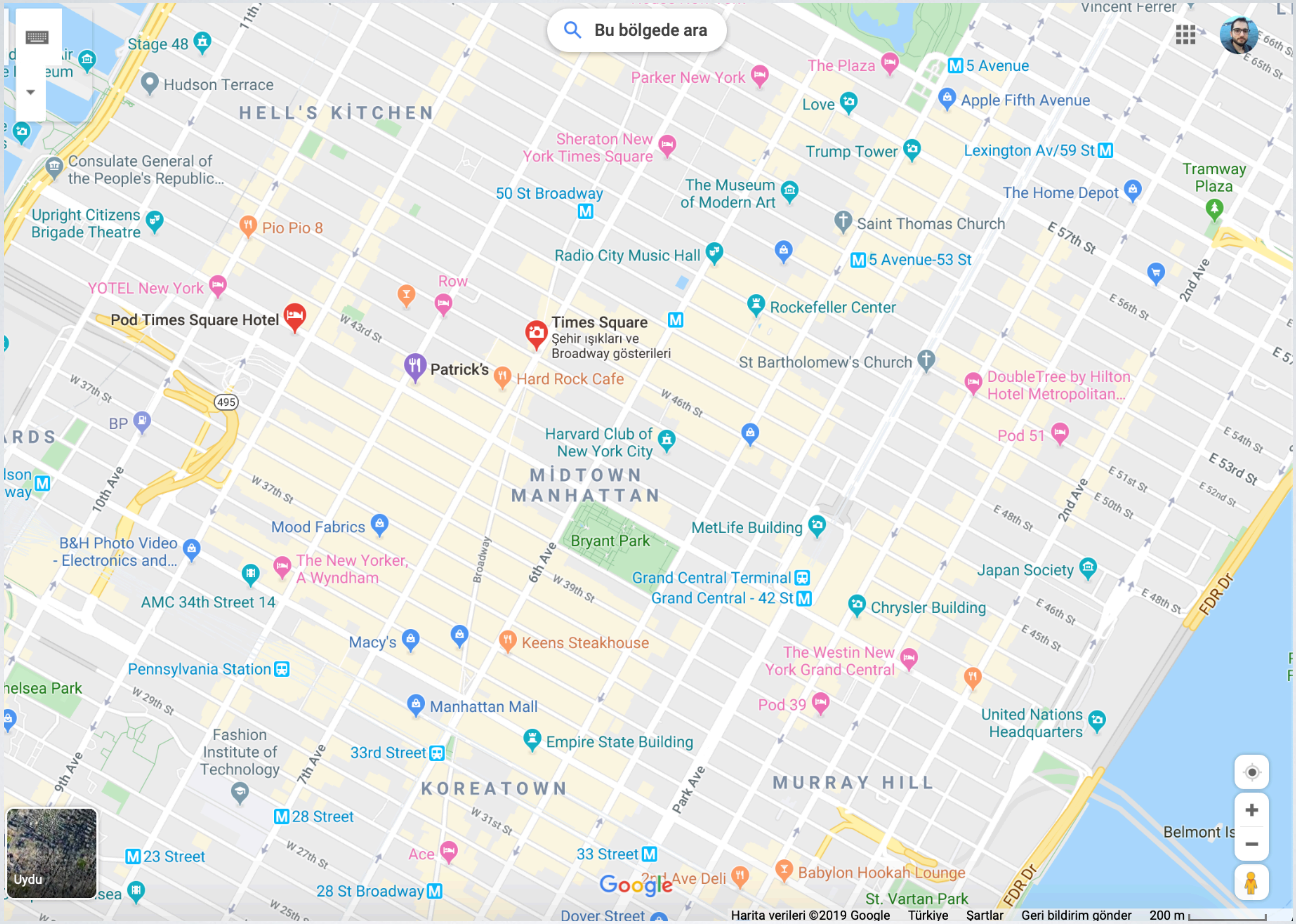
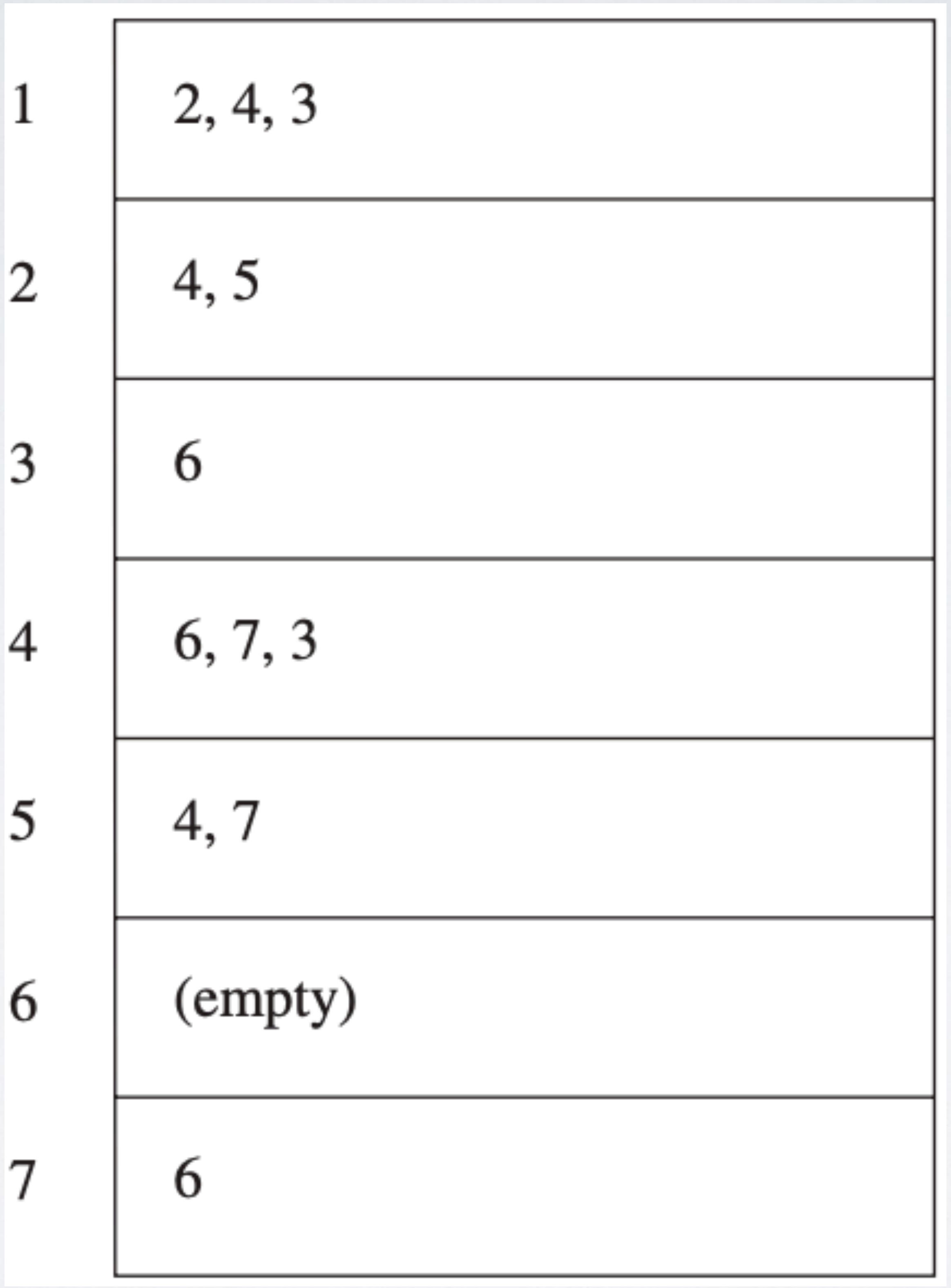


Figure: Manhattan, New York City.

Adjacency Lists

- If the graph is not dense, in other words, if the graph is **sparse**, a better solution is an **adjacency list** representation.
- For each vertex, we keep a list of all adjacent vertices. Space requirement is then O(|E| + |V|), which is linear in size of the graph.
- If the edges have weights, then this additional information is also stored in the adjacency lists.
- Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially doubles.

Sample Adjacency List



1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

Figure: An adjacency list representation of a graph.

Adjacency Lists

- There are several alternatives for maintaining the adjacency lists.
- First, observe that the lists themselves can be maintained in either vectors or lists.
- However, for **sparse** graphs, when using vectors, the programmer may need to initialize each vector with a smaller capacity than the default; otherwise, there could be significant wasted space.
- Therefore, we will use linked list in this PS.

Topological Sort

- A **topological sort** is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.
- A directed edge (v, w) indicates that course v must be completed before course w may be attempted.
- It is clear that a topological ordering is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.
- Furthermore, the ordering is not necessarily unique; any legal ordering will do.

Topological Sort

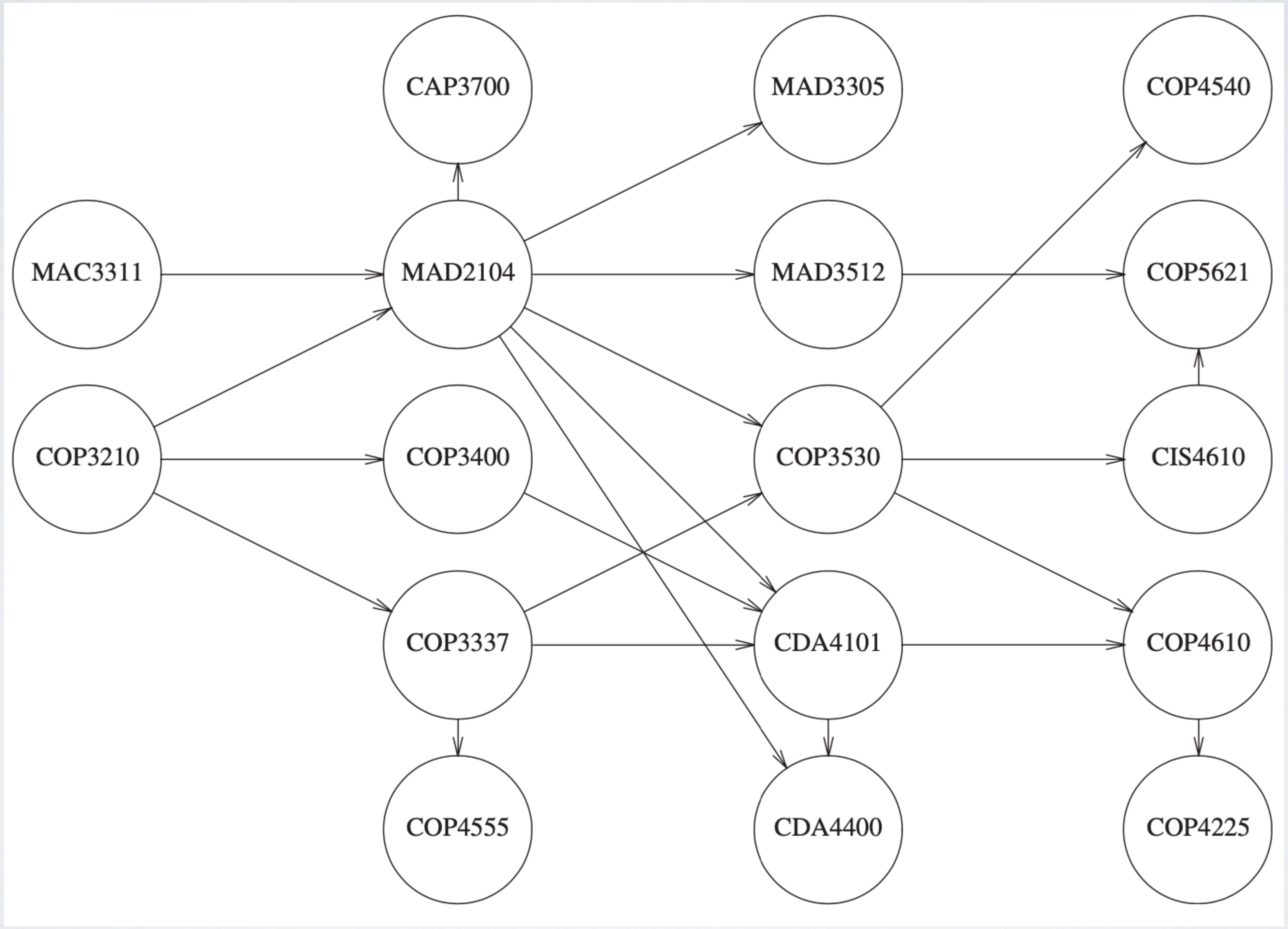
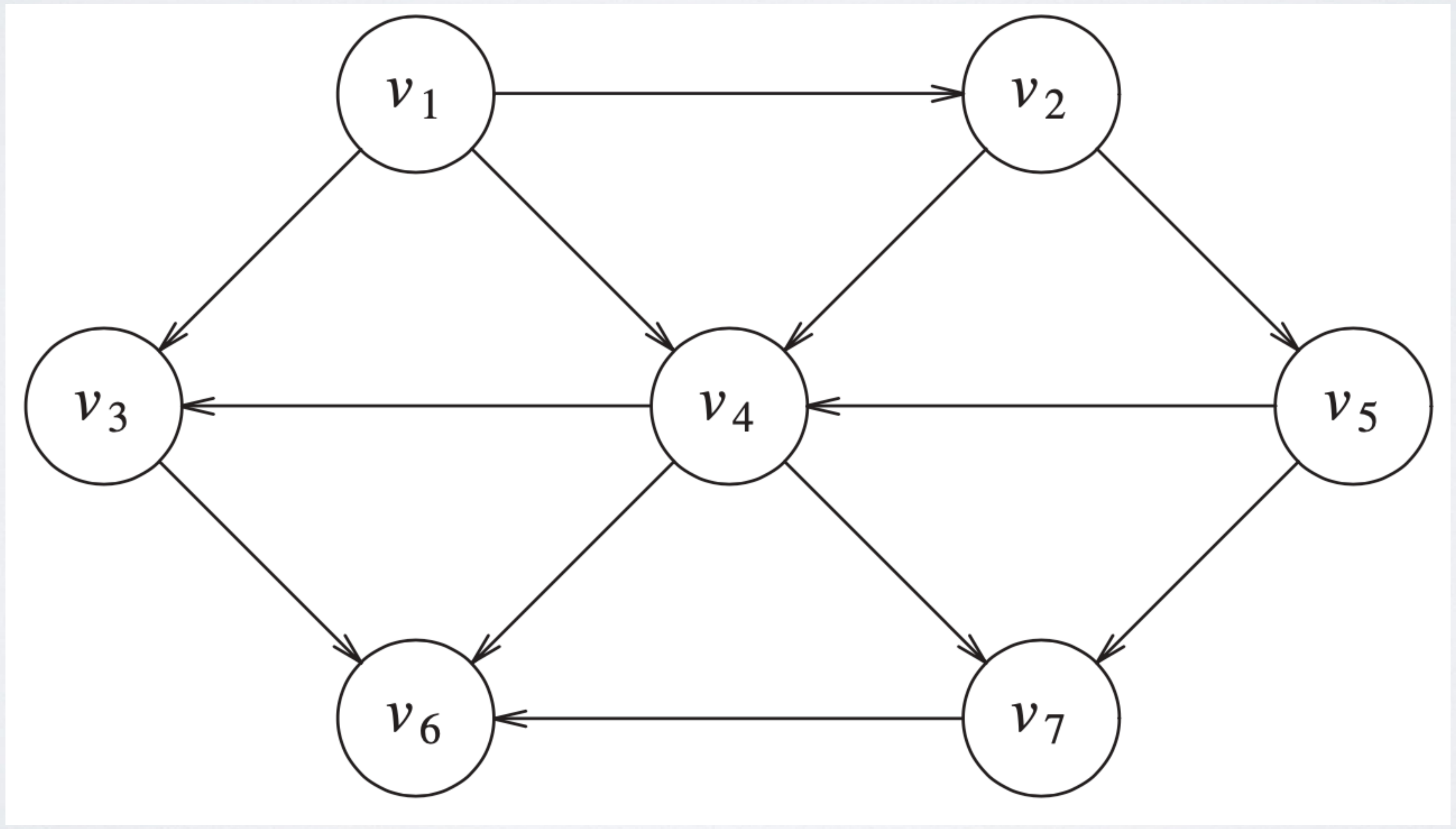


Figure: An acyclic graph representing course prerequisite structure.

Topological Sort

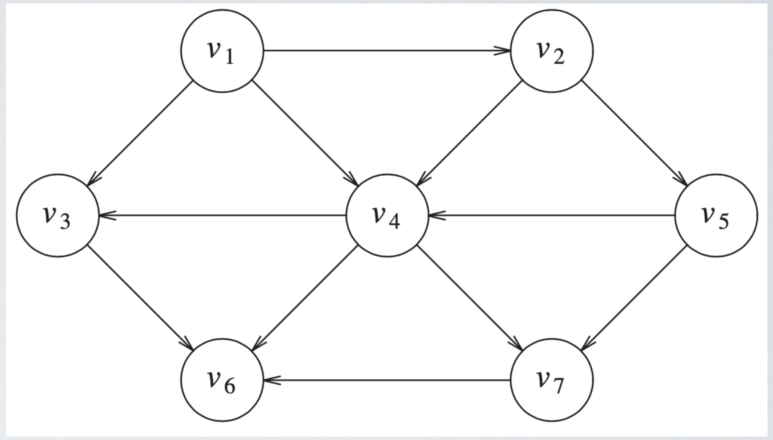
- In the graph below,
  - "v1, v2, v5, v4, v3, v7, v6" and
  - "v1, v2, v5, v4, v7, v3, v6" are both topological orderings.

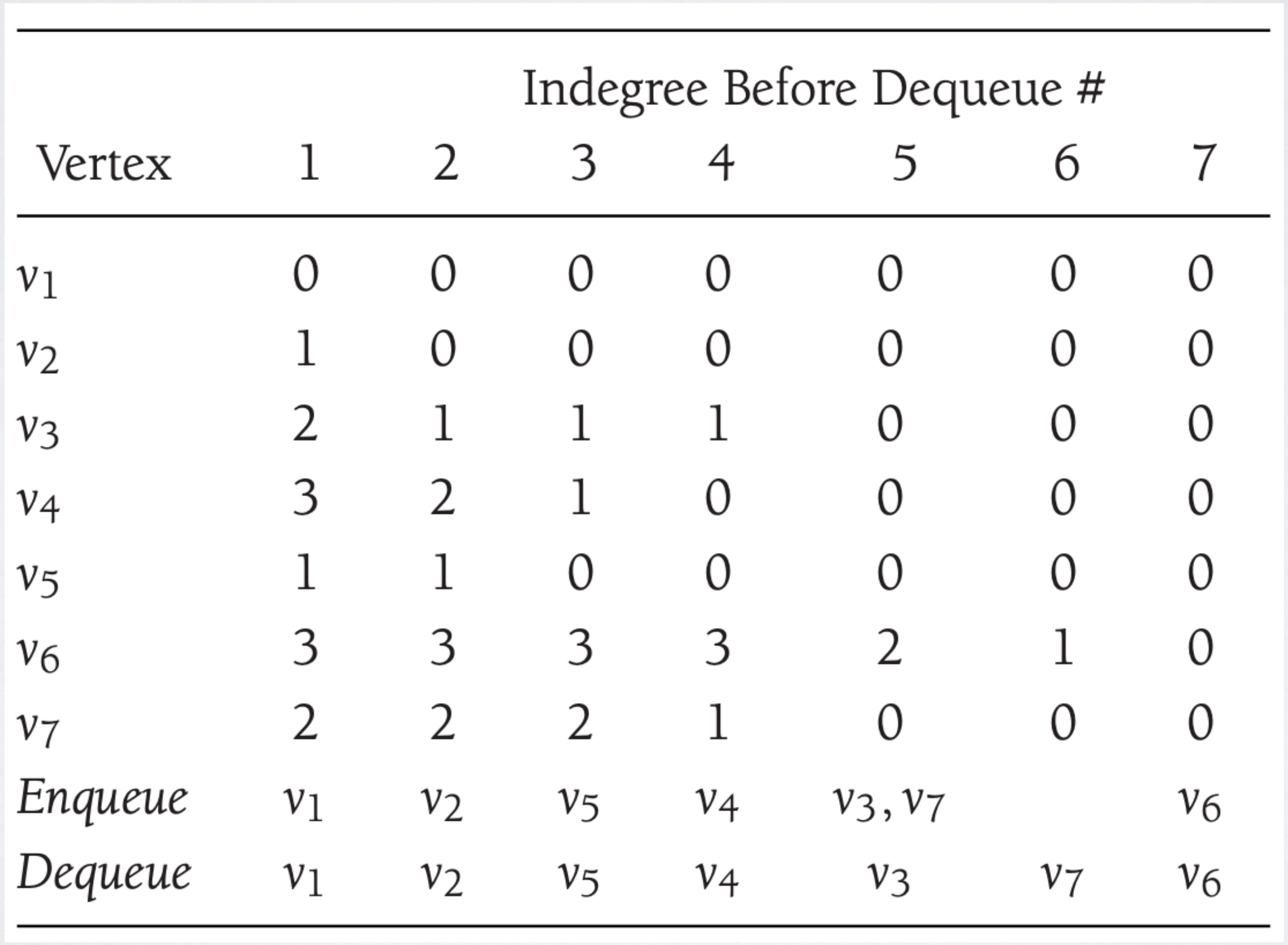


Topological Sort Algorithm

- A simple algorithm to find a topological ordering is
  - First to find any vertex with no incoming edges.
  - Then, print this vertex, and remove it, along with its edges, from the graph.
  - Finally, apply this same strategy to the rest of the graph.
- What if we can’t remove all vertices?
- To formalize this, we define the indegree of a vertex v as the number of edges (u, v). We compute the indegrees of all vertices in the graph, and store it for each vertex in the graph.

Topological Sort Example

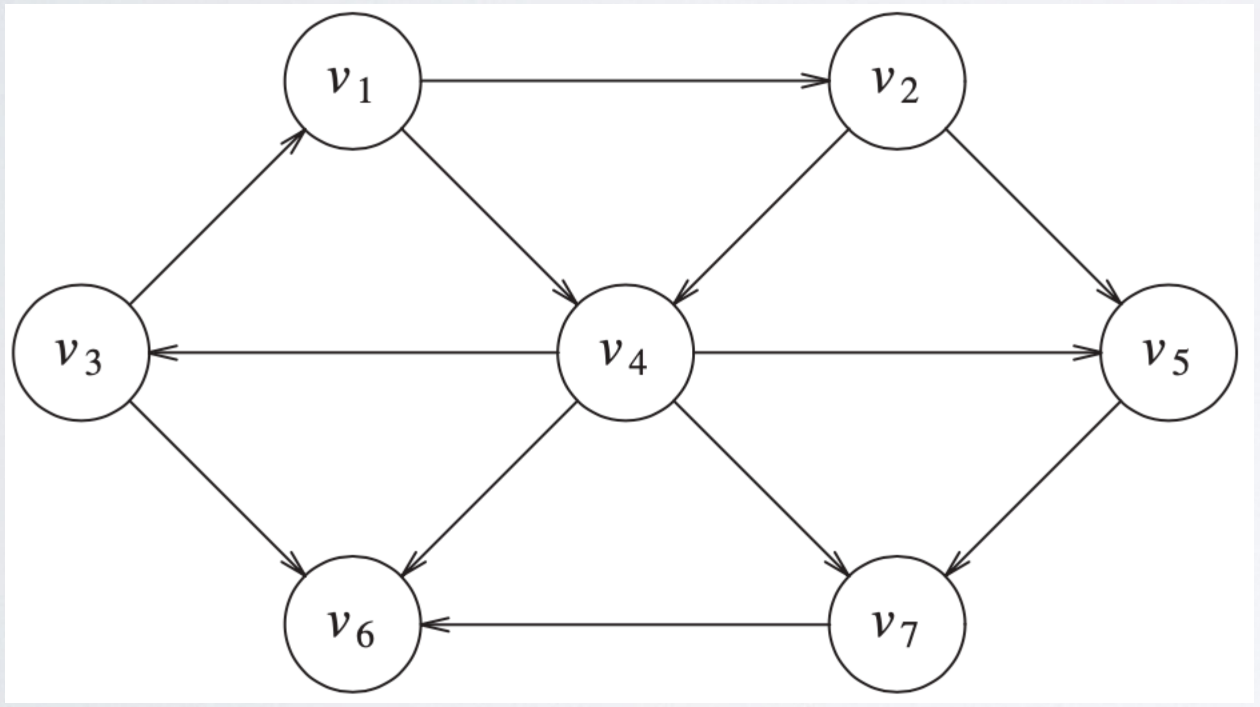


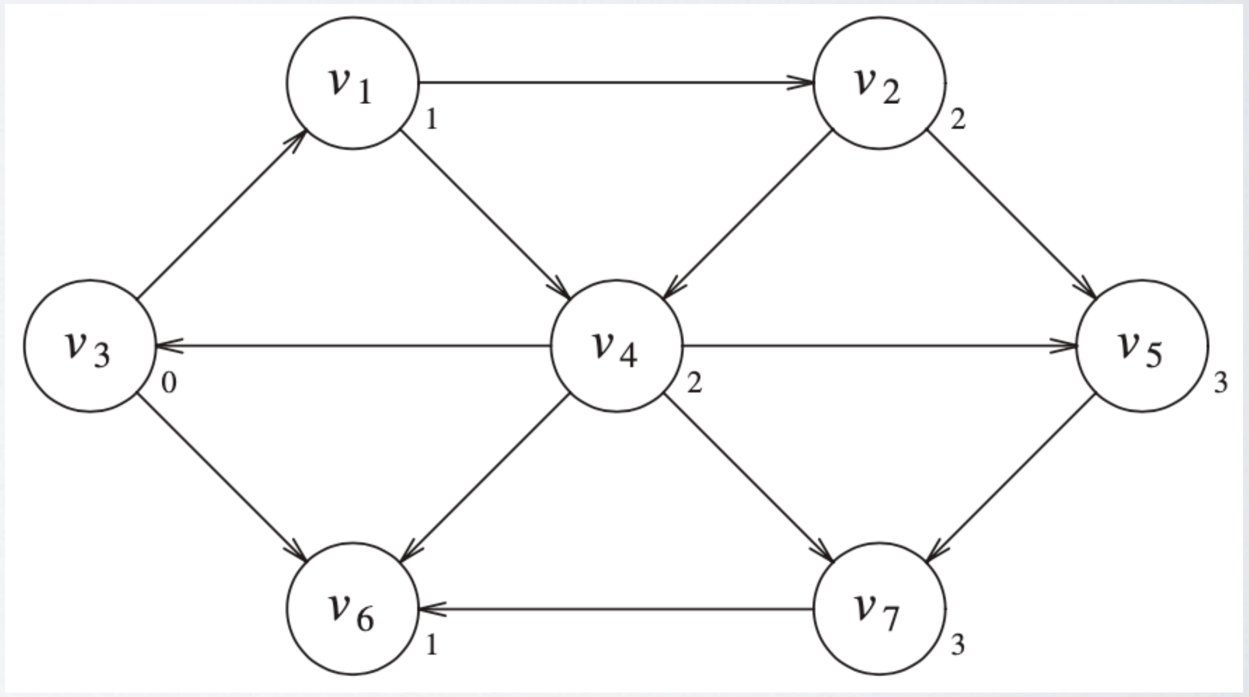


	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

Breadth First Search (BFS)

- BFS operates by processing vertices in layers:
    - Vertices closest to the start are evaluated first, and the most distant vertices are evaluated last.





BFS Algorithm

```
Push source vertex in a queue and mark it as processed
While queue is not empty
    Read vertex v from the queue
    For all neighbors of v (let’s call it w):
        If w is not processed
            Mark w as processed
            Push(enqueue) w in the queue
            Record the parent of w to be v (necessary only if we need the
            shortest path tree)
```

DEMO TIME