Course: Mathematics for IT Professionals

Lecture 10

Graphs

By

Solomon Mensah

Outline

The topics to be treated in this lecture are:

- Concept of Graphs
- Types of Graphs
- Properties of Graphs
- Adjacency Matrix vs Adjacency List
- Graph Traversal
- Applications of Graphs

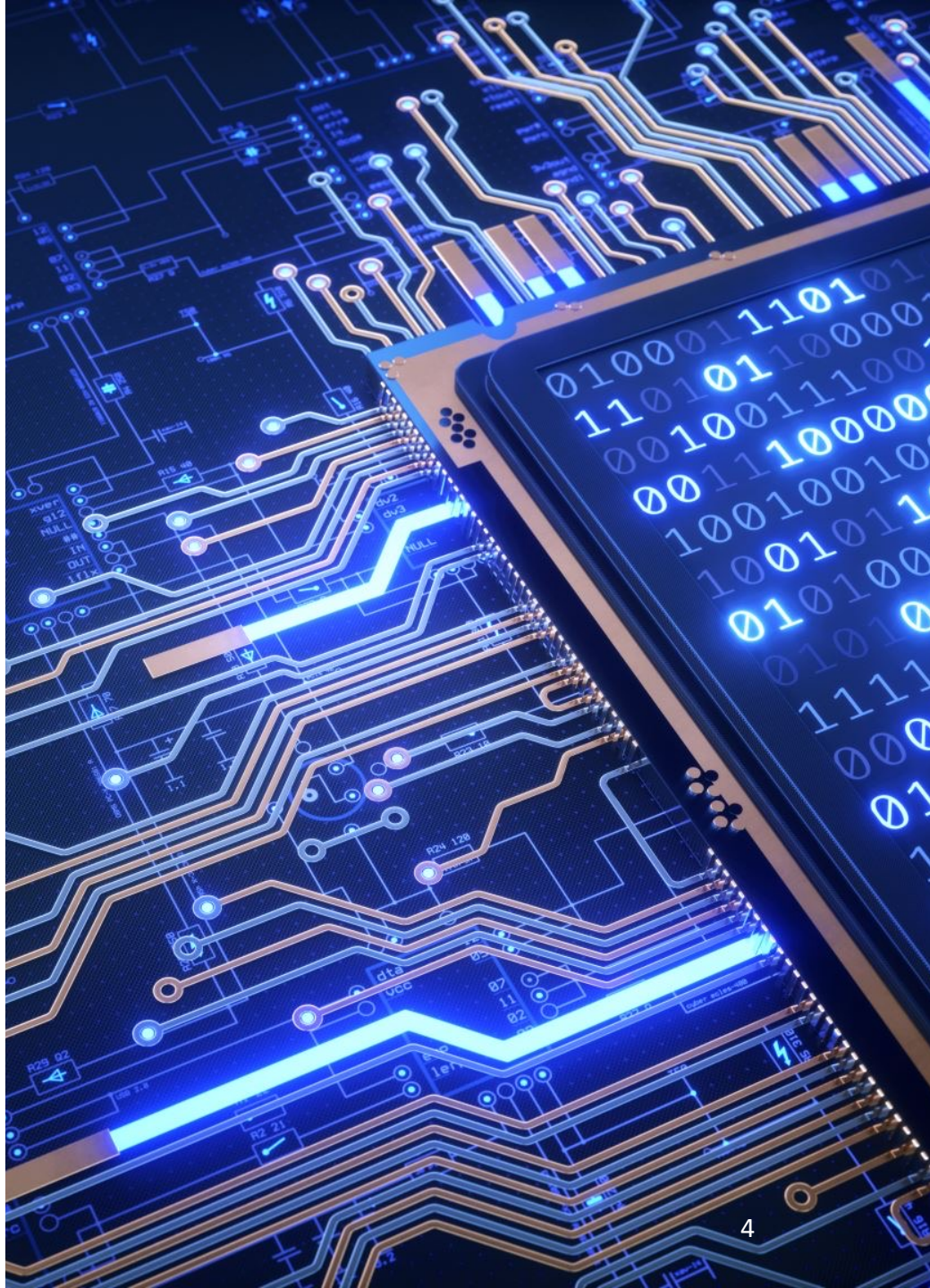
Lecture Learning Outcomes

At the end of the session, you will be able to

- understand concepts of graphs
- know some properties of graphs
- construct adjacency matrix and list based on a given graph
- know the application of graphs used to solve real-life problems

Introduction

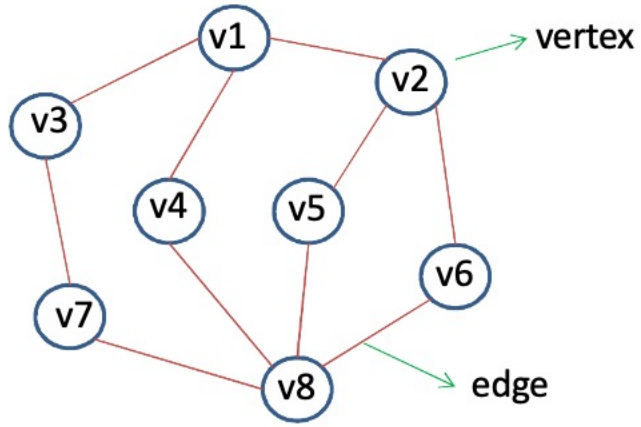
- Graph is a non-linear data structure used to model and represent a wide variety of systems in real world
- Just like a tree, a graph also consists of a set of nodes (or vertices) and edges
- But, unlike the trees, graphs will have no restrictions with respect to the connections between the nodes. This means nodes can be connected in any possible way.
- A tree in fact can be treated as a special kind of graph

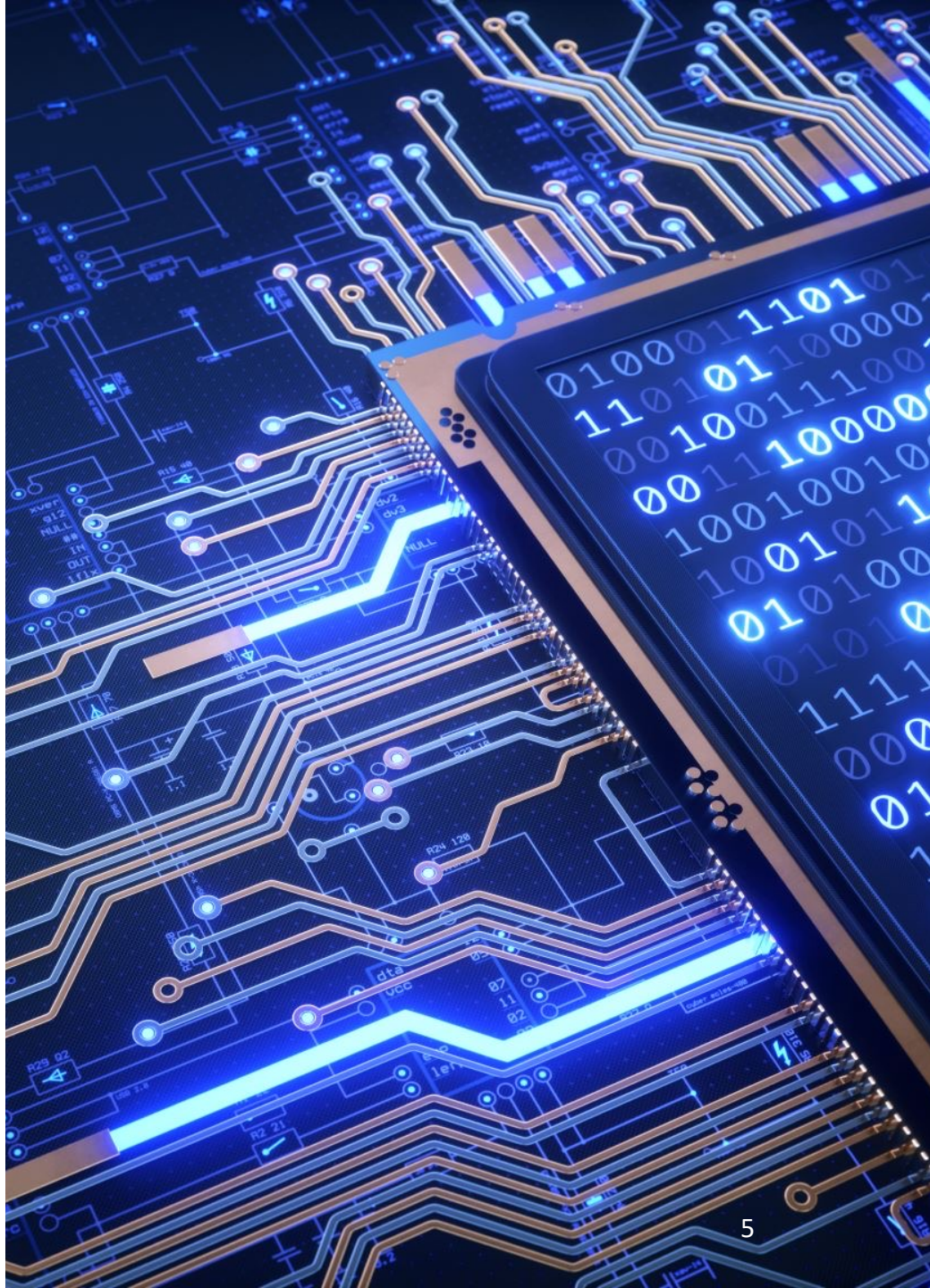


Rosen, K. H. (2012). *Discrete mathematics and its applications (7th Edition)*. McGraw-Hill.

Graph: Definition

- A **graph** *G* is an *ordered pair* of a set of **vertices**, *V*, and a set of **edges** or **arcs**, *E*.
- That is, graph $G = (V, E)$
- Each edge can be an *ordered*
- (or *directed*) or unordered
- (or undirected) pair of vertices
- Ordered pair of two vertices
- u, v is written as (u, v)
- Unordered pair is written as $\{u, v\}$ or $\{v, u\}$
- In a directed edge (u, v), u is called **origin** and v is called **destination**

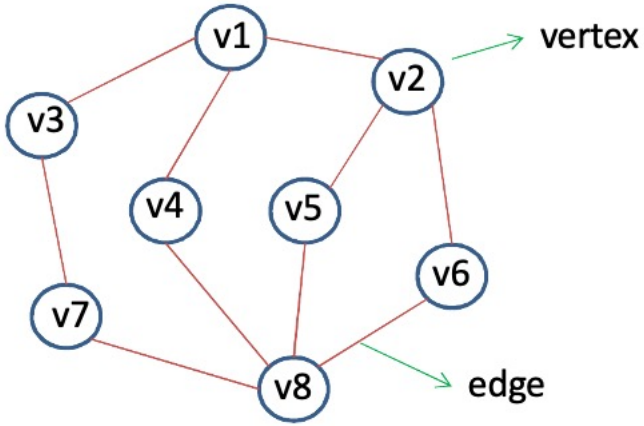




Rosen, K. H. (2012). *Discrete mathematics and its applications (7th Edition)*. McGraw-Hill.

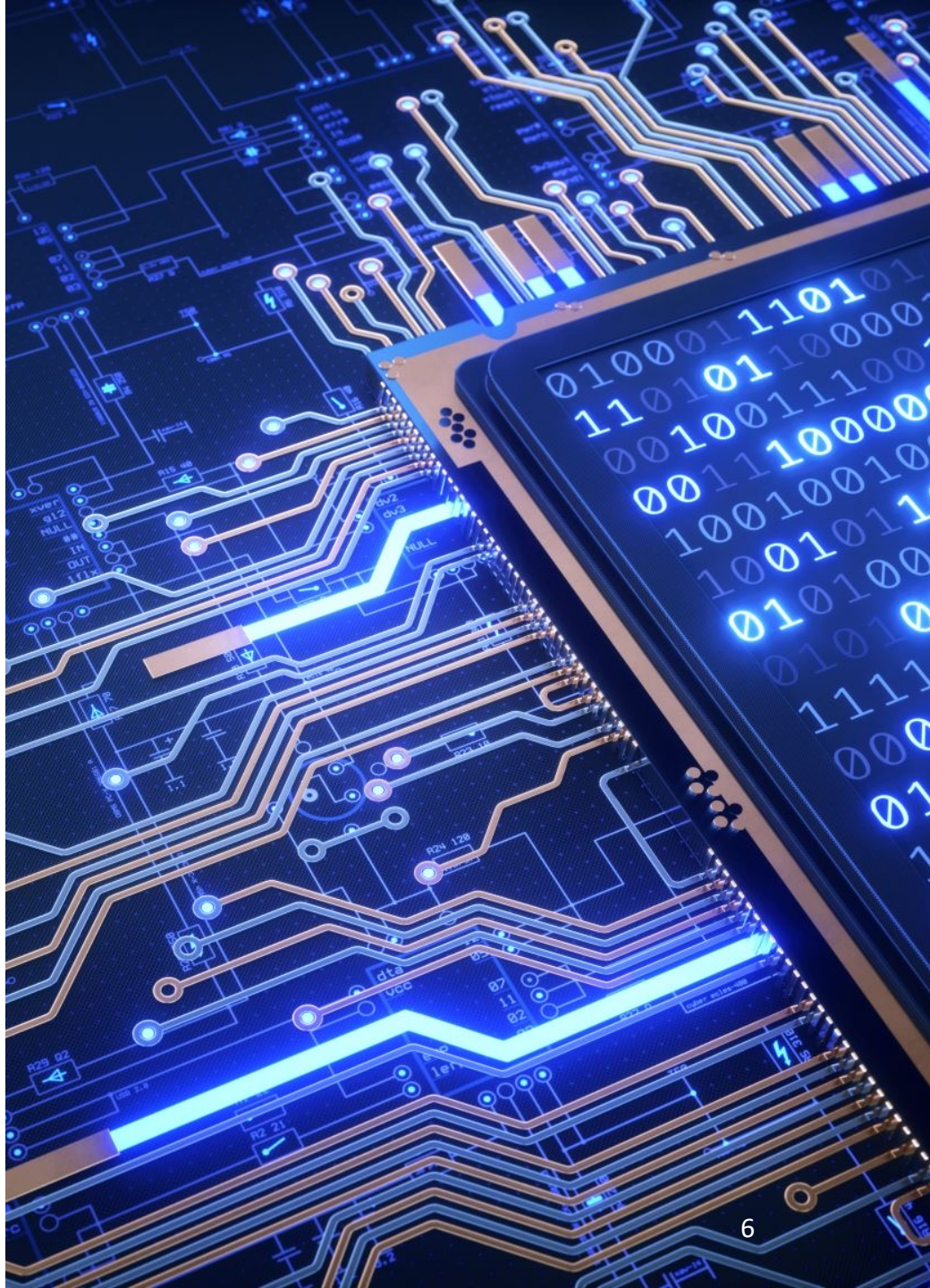
Graph: Definition

- Example: For the graph shown,
  - V = {v1, v2, v3, v4, v5, v6, v7, v8}
  - E = { {v1, v2}, {v1, v3}, {v1, v4},
    - {v2, v5}, {v2, v6}, {v3, v7},
    - {v4, v8}, {v7, v8}, {v5, v8},
    - {v6, v8} }



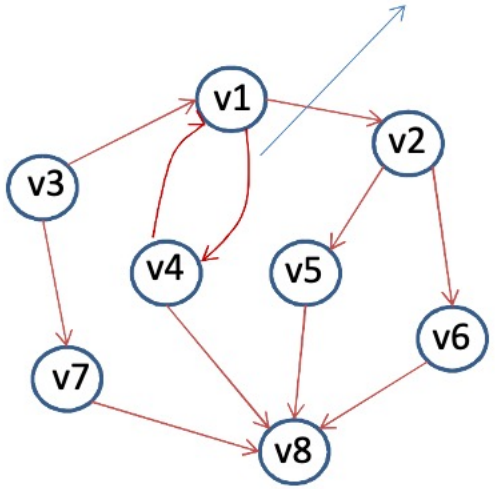
Unordered pairs. Hence represented with braces

Note: A graph may contain both directed and undirected edges

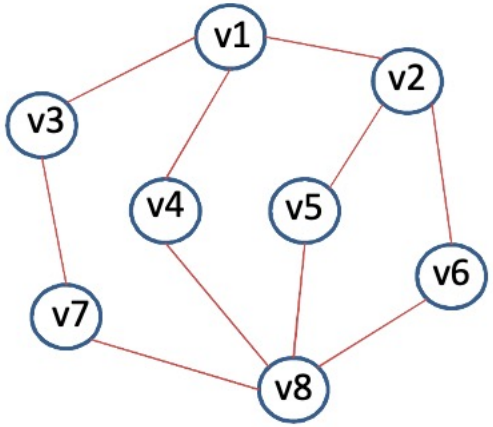


Graphs: Types

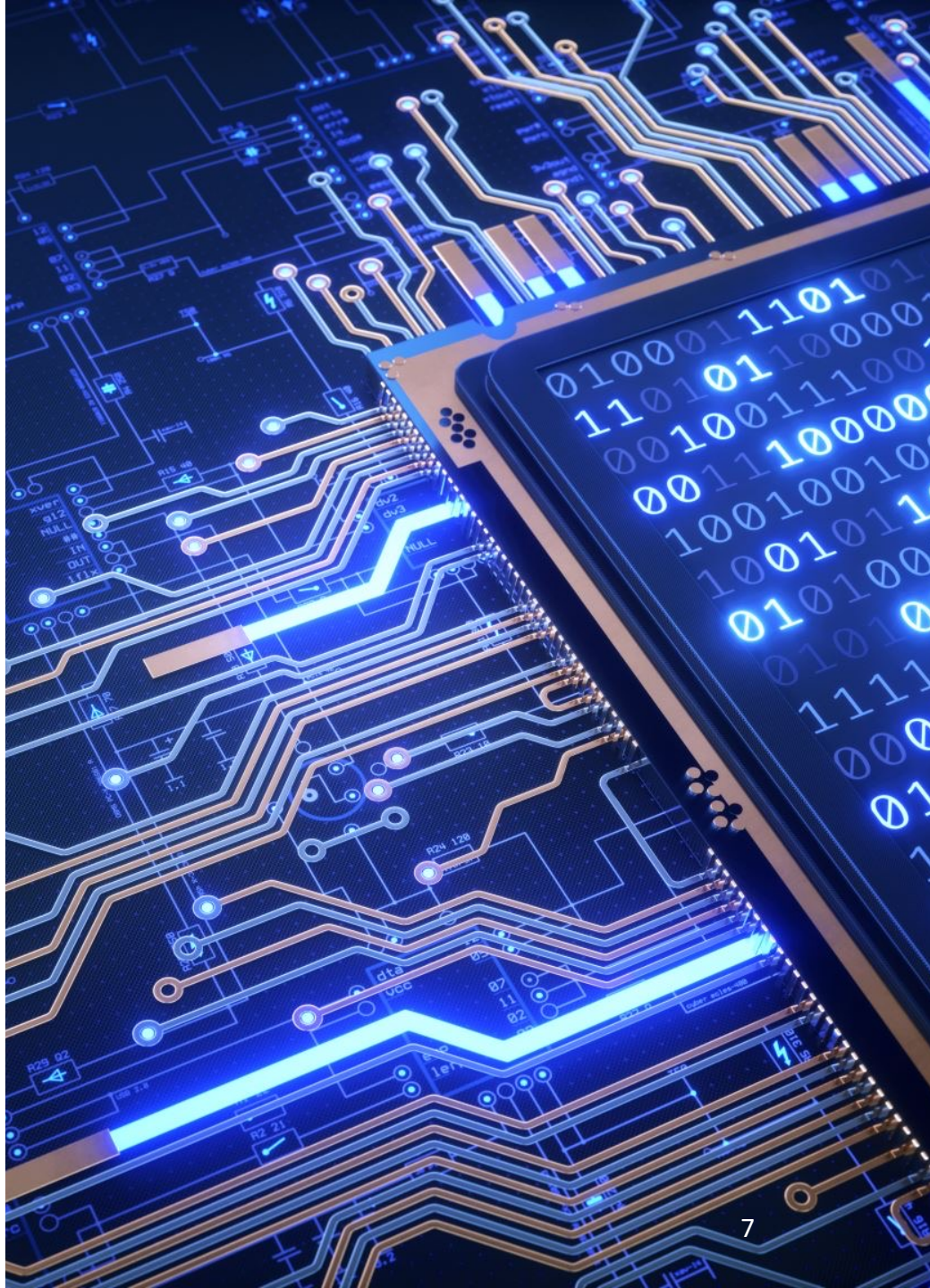
- The edge (v1, v4) is different from (v4, v1)



Directed graph or ***Digraph***
Edges are unidirectional

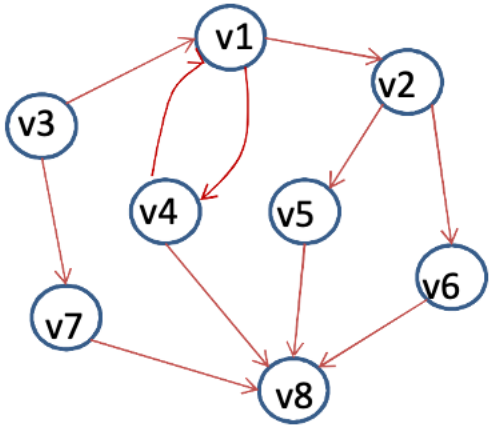


Undirected graph
Edges are bi-directional



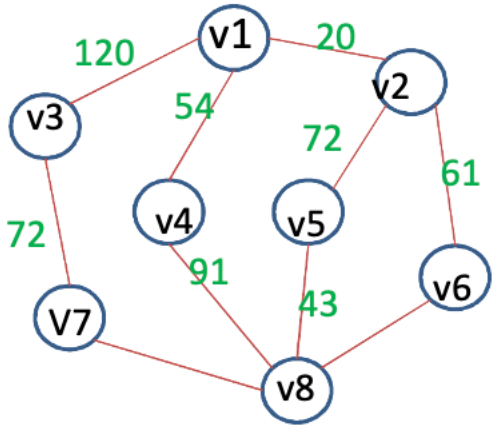
Graphs: Types

- An edge may have a value associated with it called ***weight*** or ***cost***.
- Accordingly, a graph may be ***weighted*** or ***unweighted***

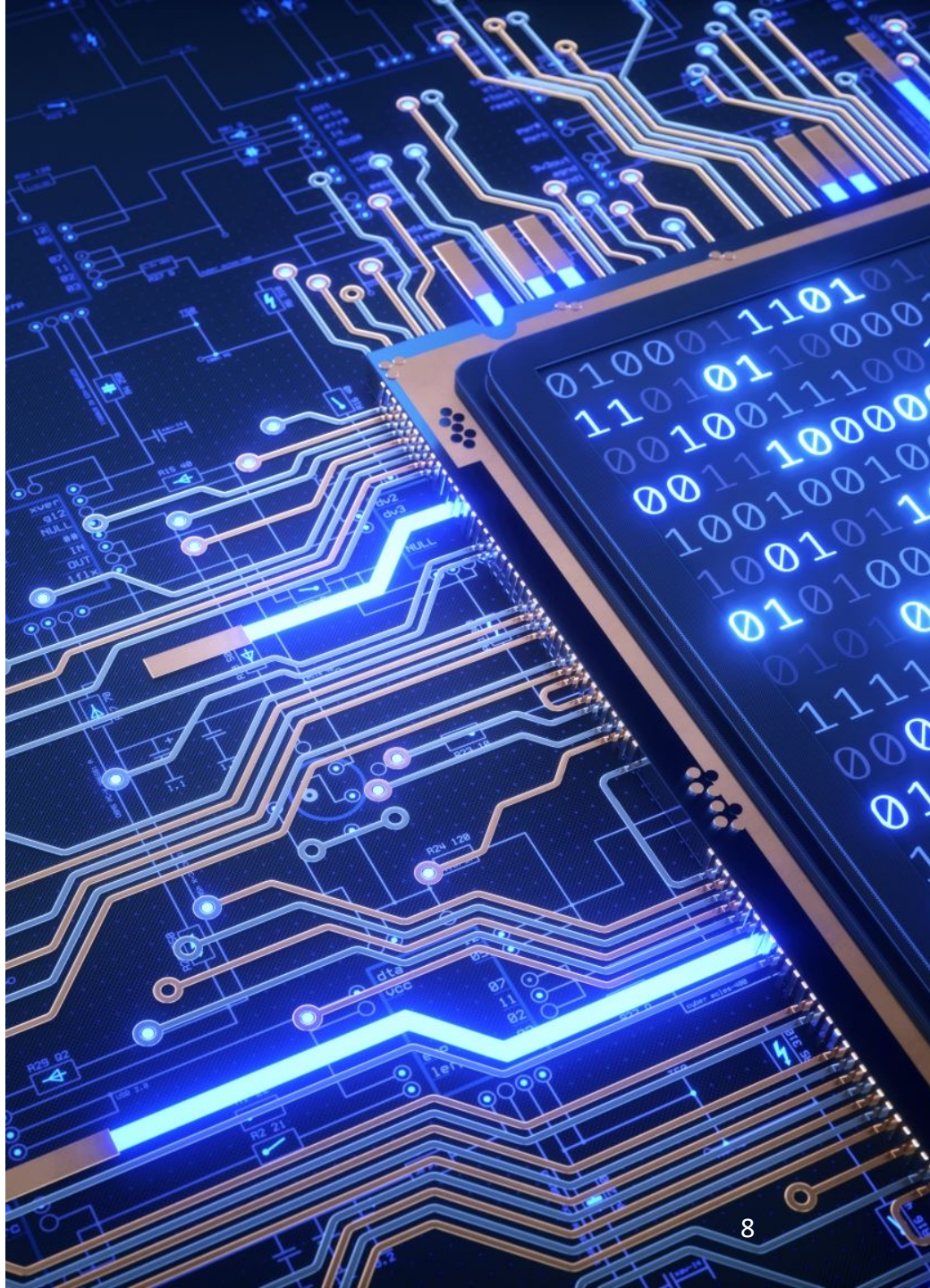


Unweighted digraph

Can be seen as weighted graph with all edges having same weight (we assume 1)

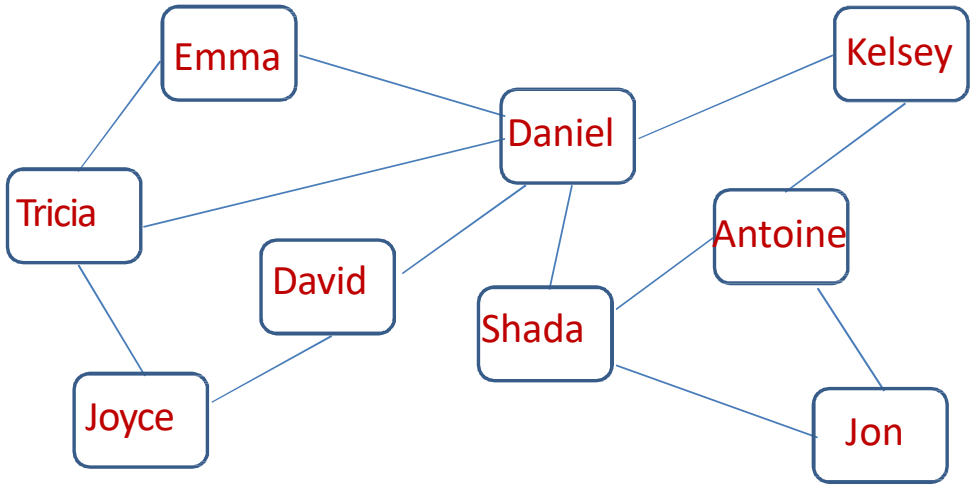


Weighted undirected graph



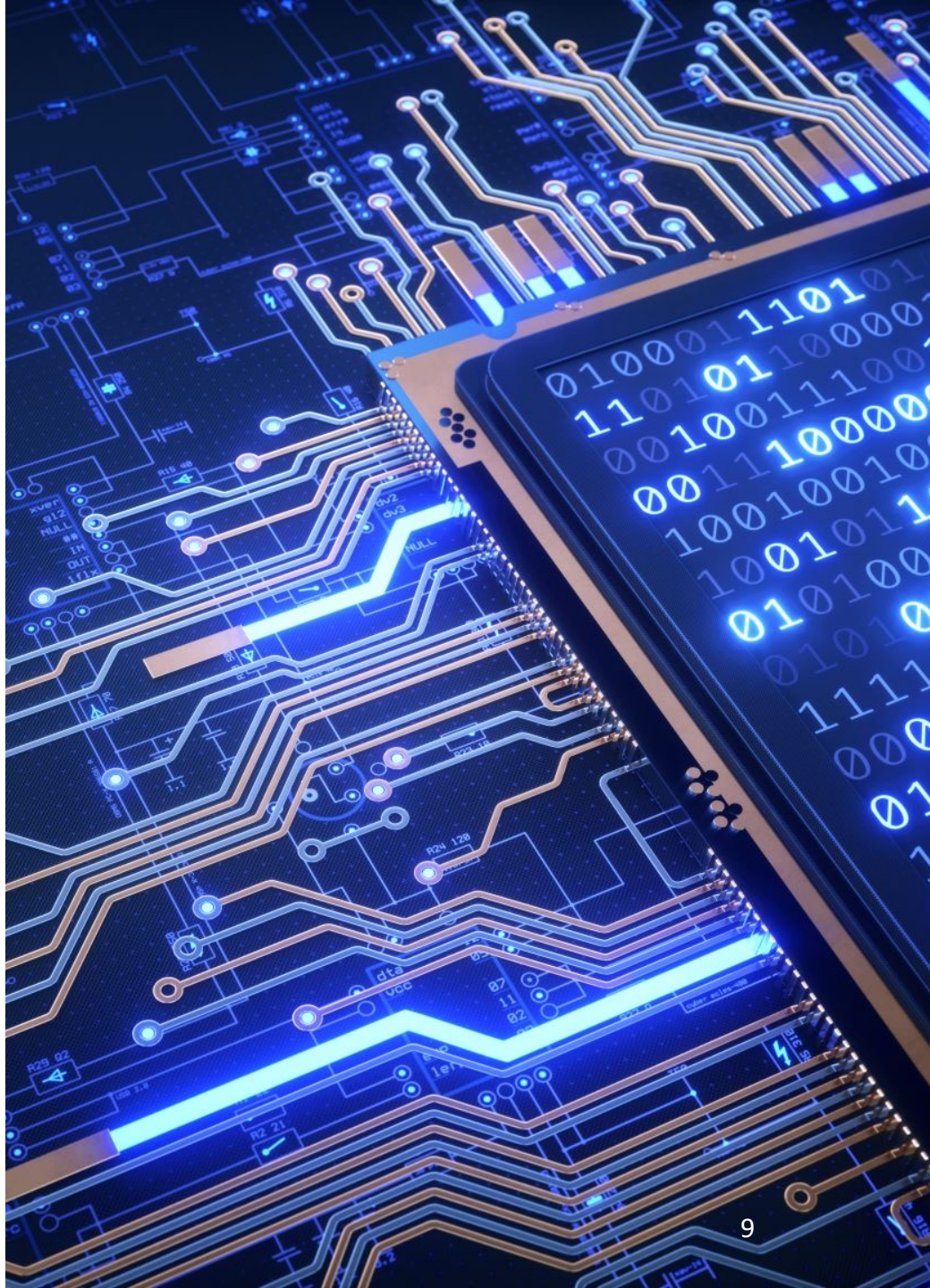
Application Example 1

- Graphs can be used to represent or model any collection of objects which have pairwise relationships
- Example: Social network like Facebook can be modeled as an
- *undirected graph*



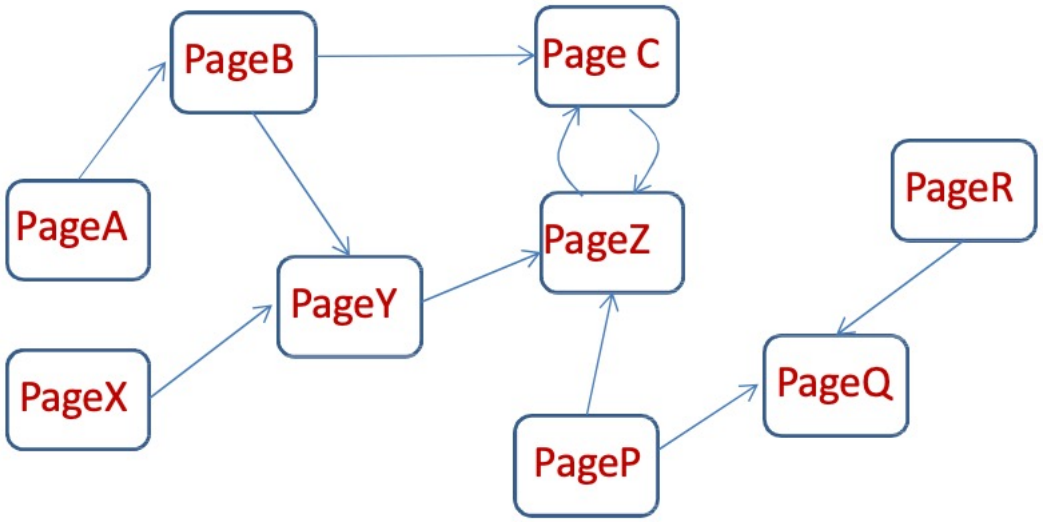
– Once modeled as graph, several problems can be solved using standard algorithms

e.g. suggesting friends (find path with length 2)



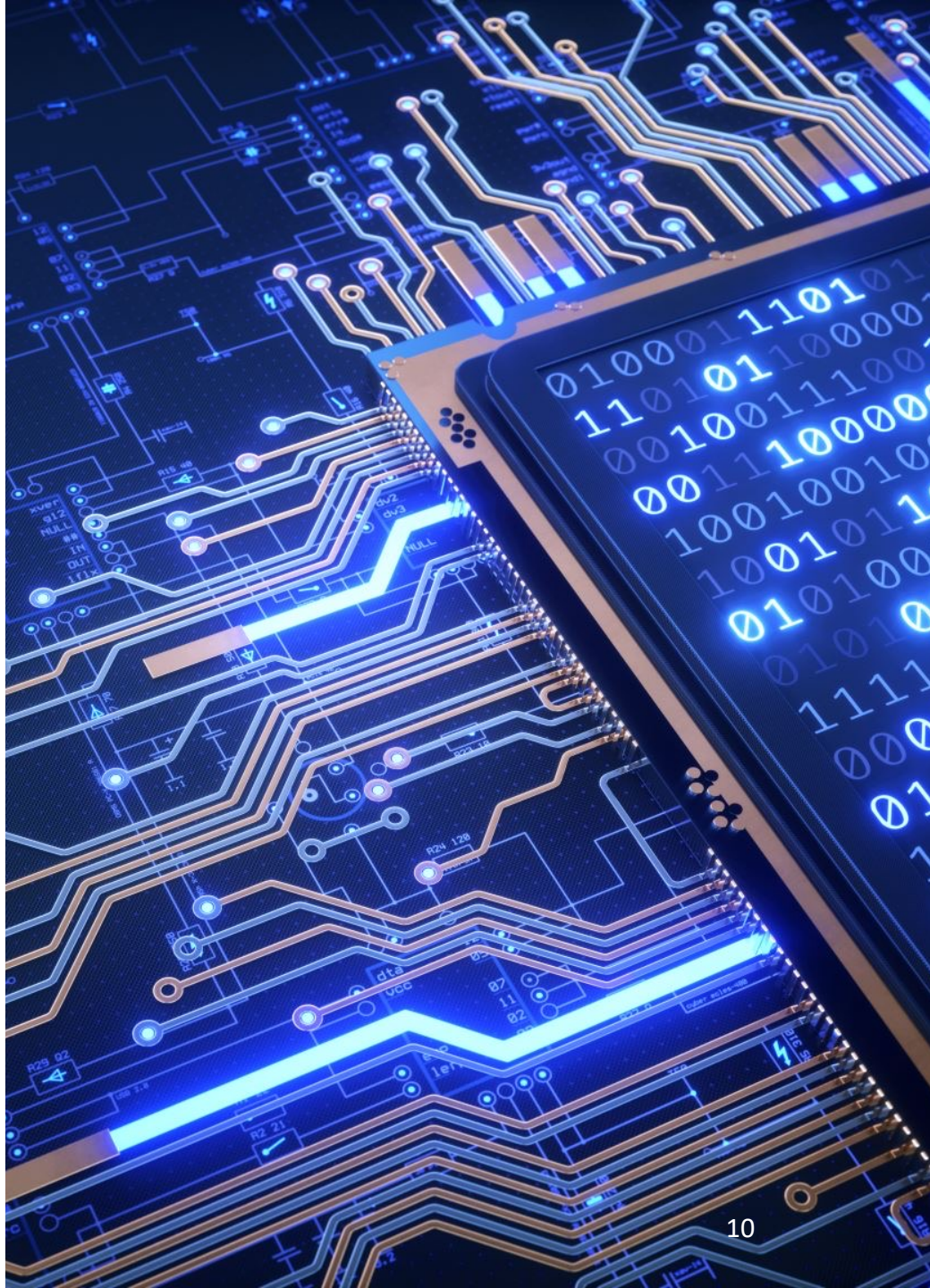
Application Example 2

- Example: Web pages on www can be represented as directed graph



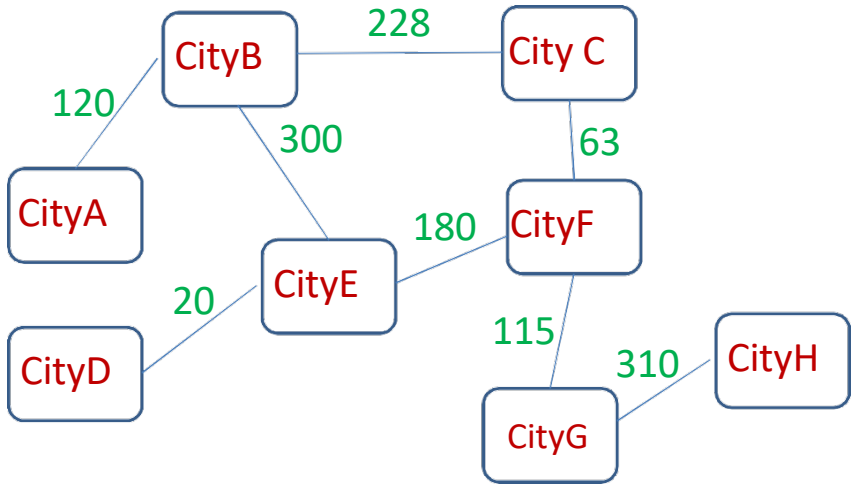
– Once modeled as graph, several problems can be solved using standard algorithms

e.g. web crawling
(traversal algorithm)



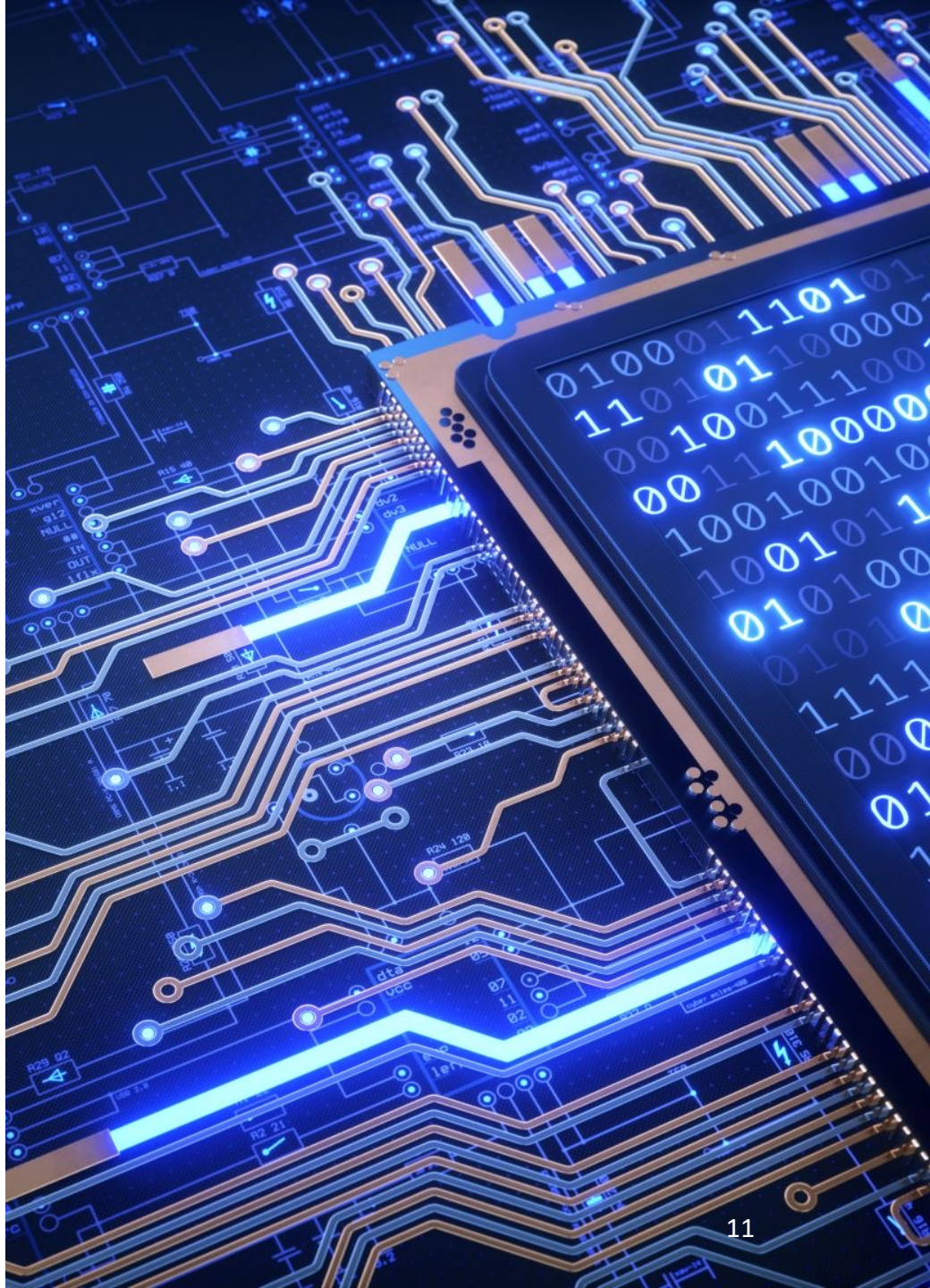
Application Example 3

- Example: Intercity road map can be modeled as undirected weighted graph



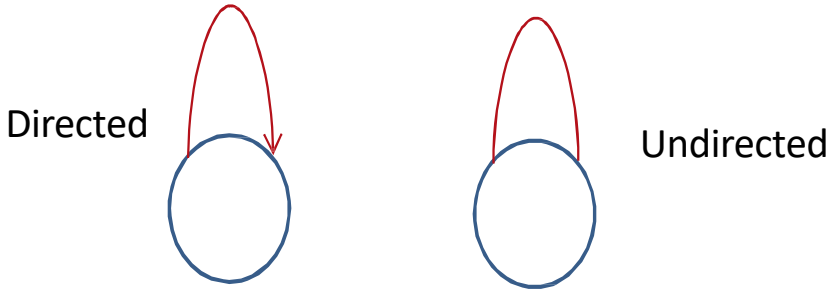
  - Once modeled as graph, several problems can be solved using standard algorithms

    e.g. best route between two cities (shortest path algorithm)

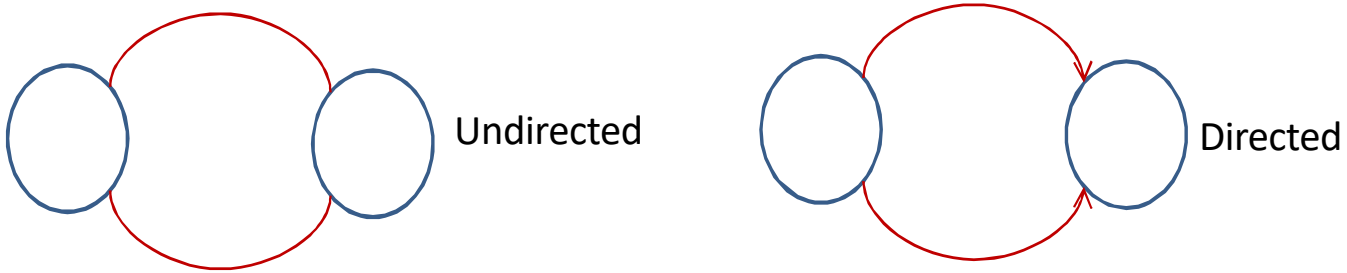


Properties of Graphs

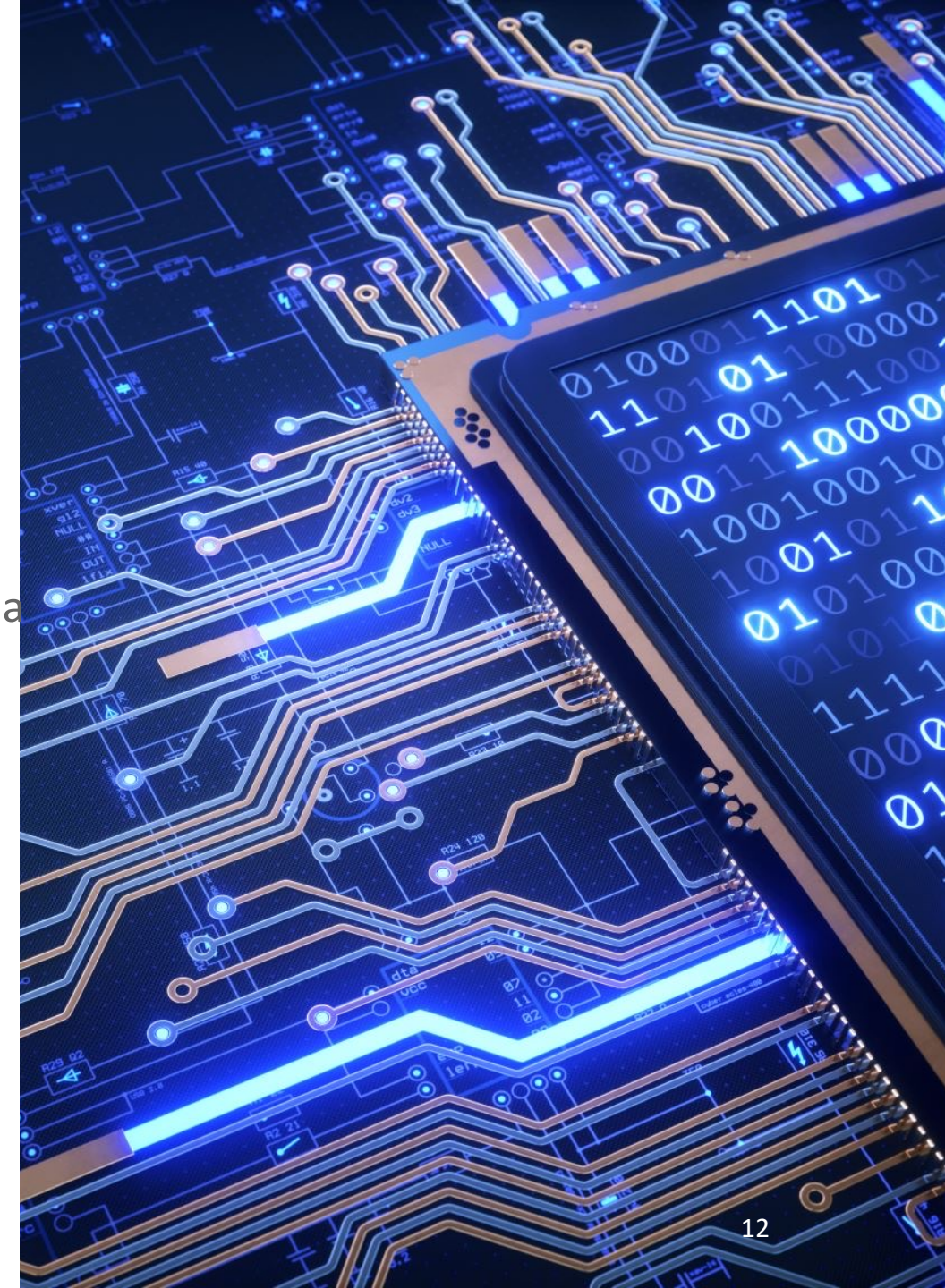
- *Self loop*: An edge is called self loop or self edge if it involves only one vertex. i.e., the origin and destination is the same node



- Example: Graph representing Internet web pages, a page can have a self link
- *Multi-edge* (or parallel edge): It is an edge that occurs more than once in a graph



- Example: Graph representing flight network between cities. A pair of cities can have multiple flights (edges) running between them



Properties of Graphs

- Self loops and multi-edges make a graph complicated to work with
- A graph with no self loop or multi-edge is called a ***simple graph***

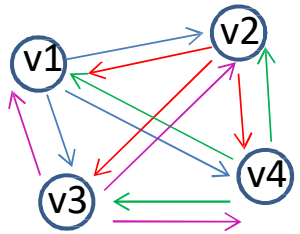
- *Number of edges*:
- Given a number of vertices in a *simple graph*, what would be the maximum possible number of edges?

Example: V = {v1, v2, v3, v4},

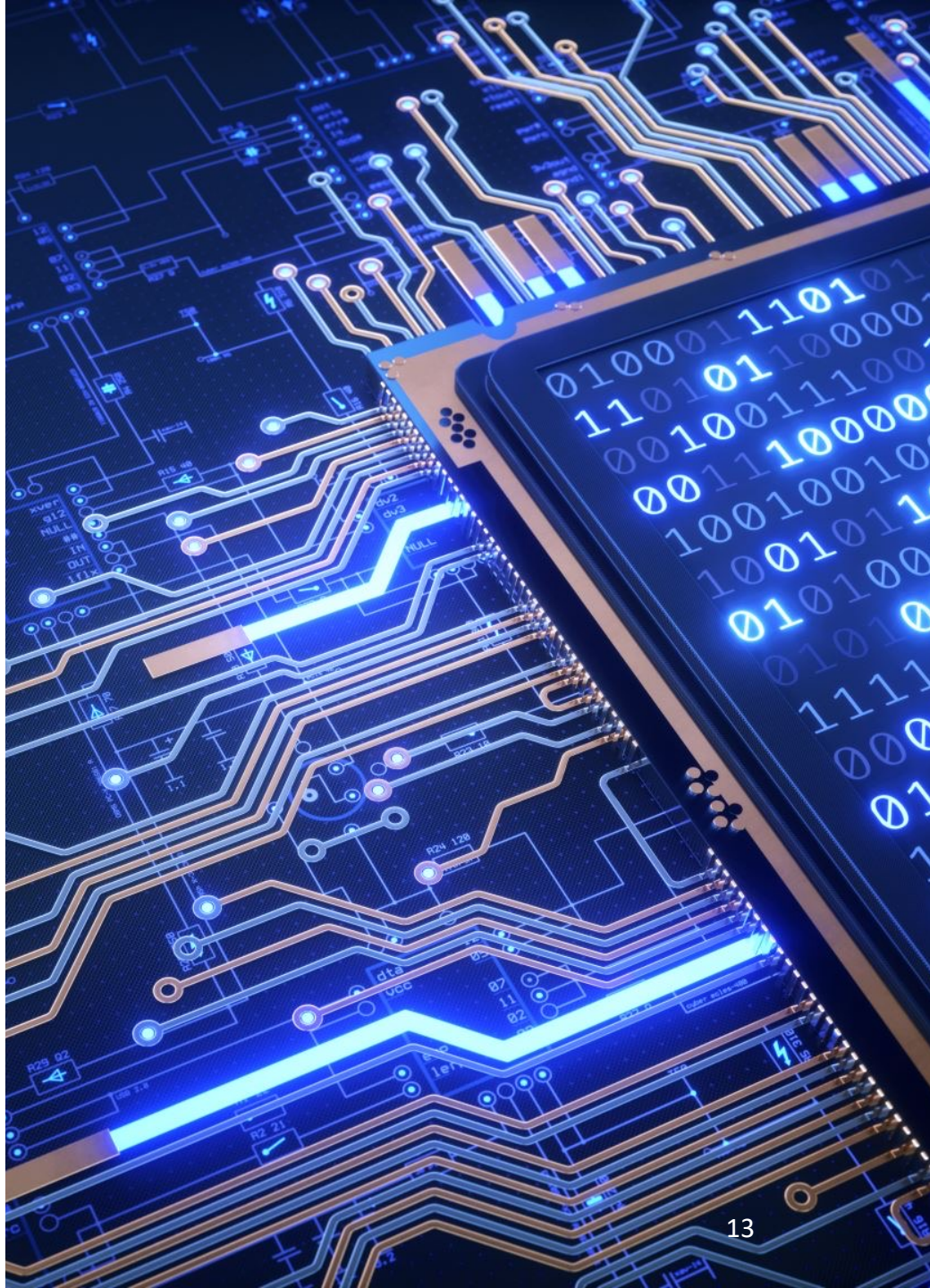
In general, If |V| = n

Then $0 \leq |E| \leq n(n-1)$, if directed graph

$0 \leq |E| \leq \frac{n(n-1)}{2}$, if undirected graph

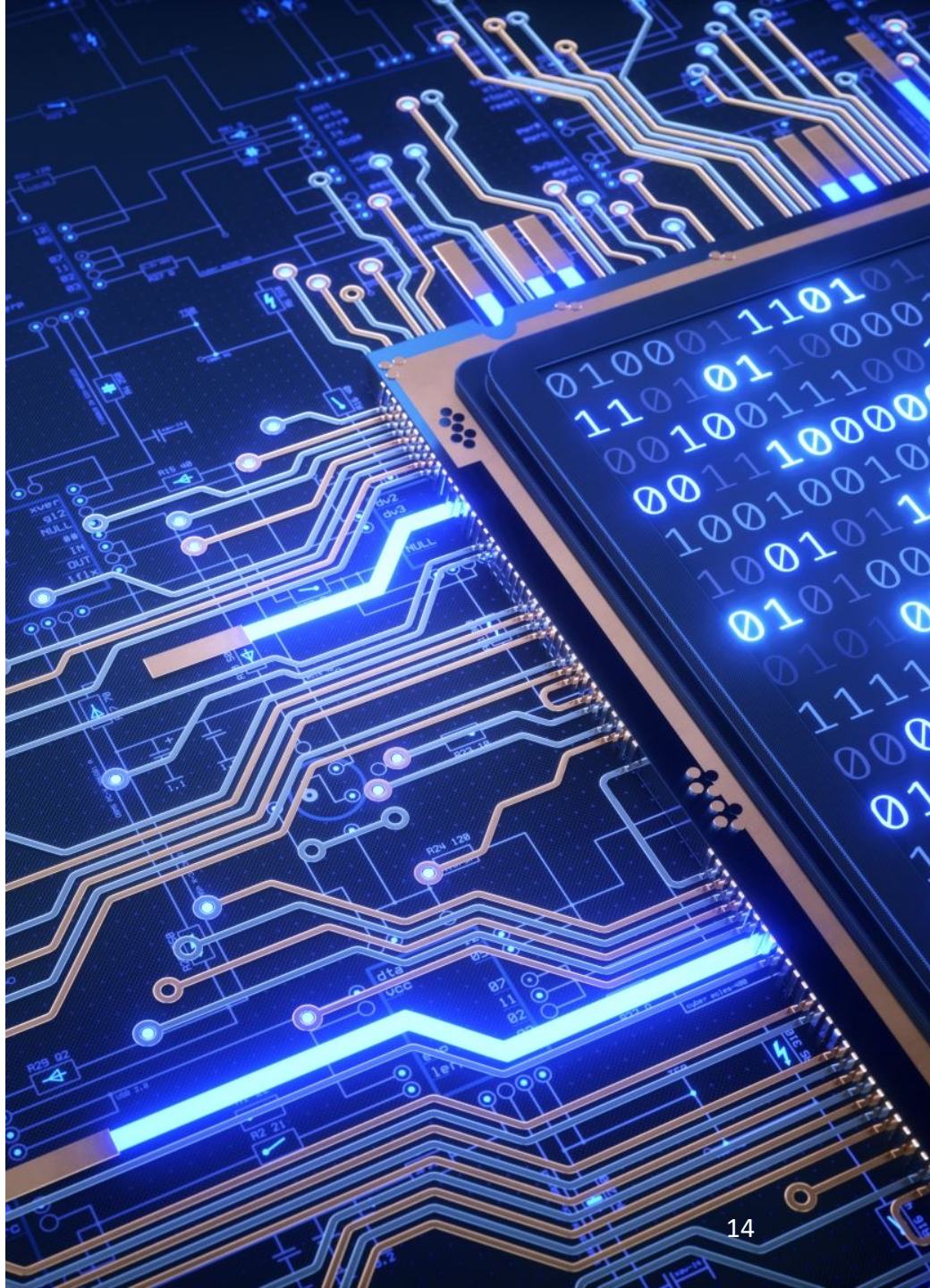


Each vertex can have an edge to all other vertices



Properties of Graphs

- A graph is called ***dense graph*** if the number of edges in it is close to maximum number of edges (i.e., too many edges)
- A graph is called ***sparse graph*** if the number of edges in it is close to the number of vertices (i.e., few edges)
- The data structure chosen to implement a graph depends on whether it is dense or sparse.
- – For example, a dense graph is represented with an ***adjacency matrix*** and a sparse graph is represented with ***adjacency list***

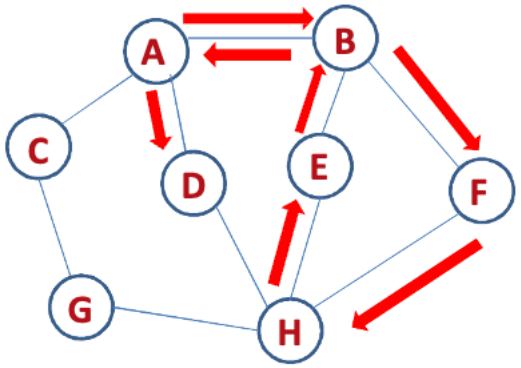


Properties of Graphs

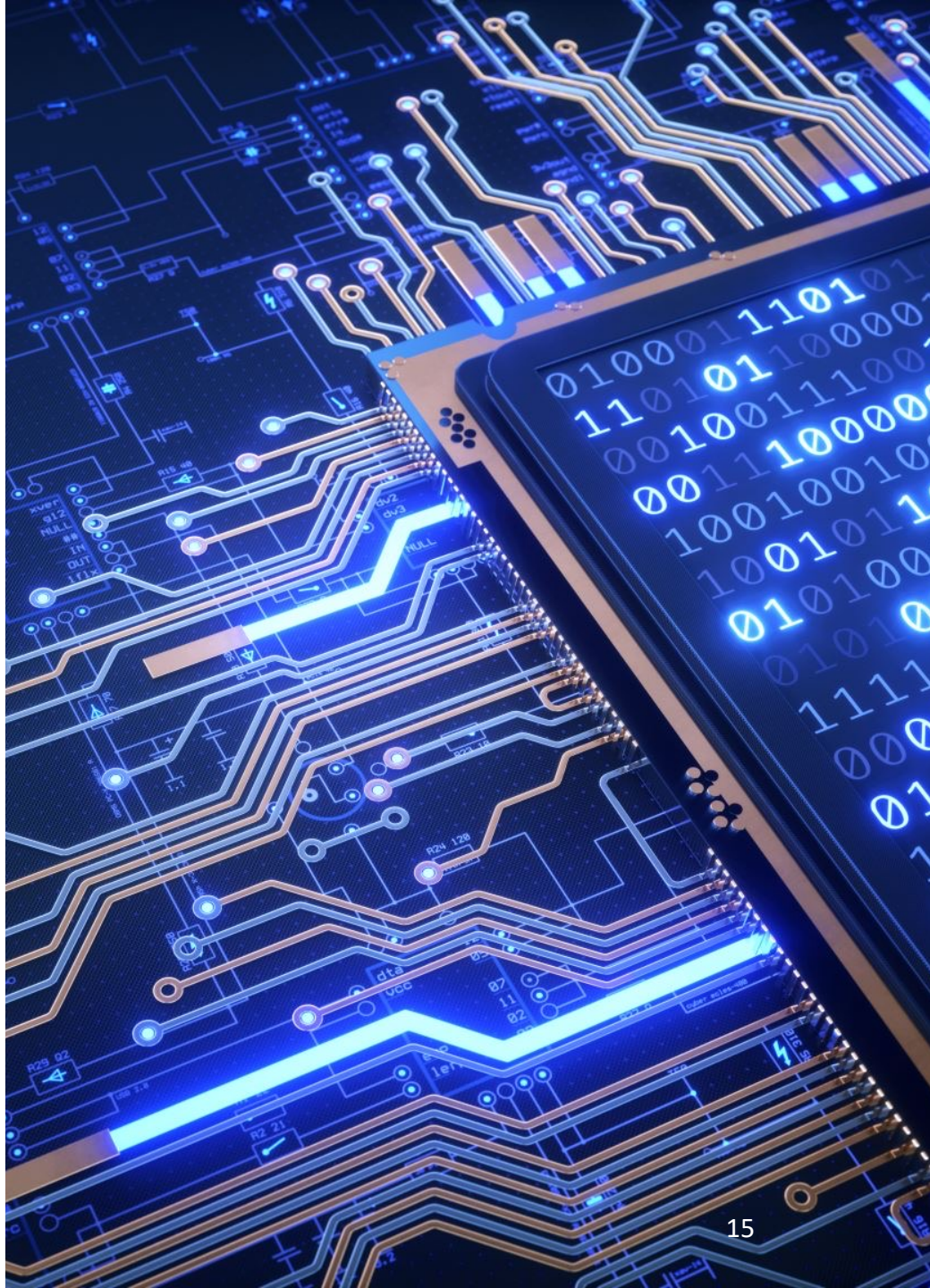
- A path in which no vertices (and thus edges) are repeated is called a ***simple path***

- Example: <A, B, F, H> is a simple path

  <A, B, F, H, E, B, A, D> is not a simple path

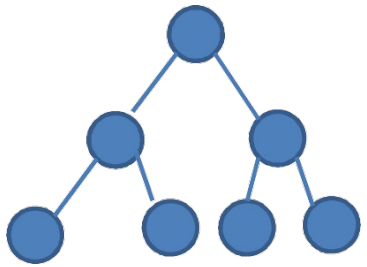


- The term ***walk*** is used to denote path in general, and the term ***path*** indicates a simple path.
- Thus, a *walk* in which no vertices are repeated is called a *path*, and that in which vertices are repeated is called a ***Trail***

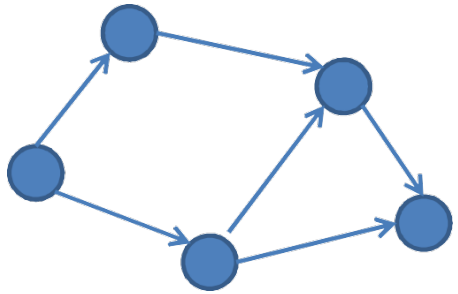


Properties of Graphs

- A walk is called ***closed walk*** if it starts and ends at same vertex and the length of walk (number of edges) is greater than 0.
  - It is sometimes called a ***cycle***.
- A graph with no cycle is called ***acyclic graph***
- A tree is an acyclic graph

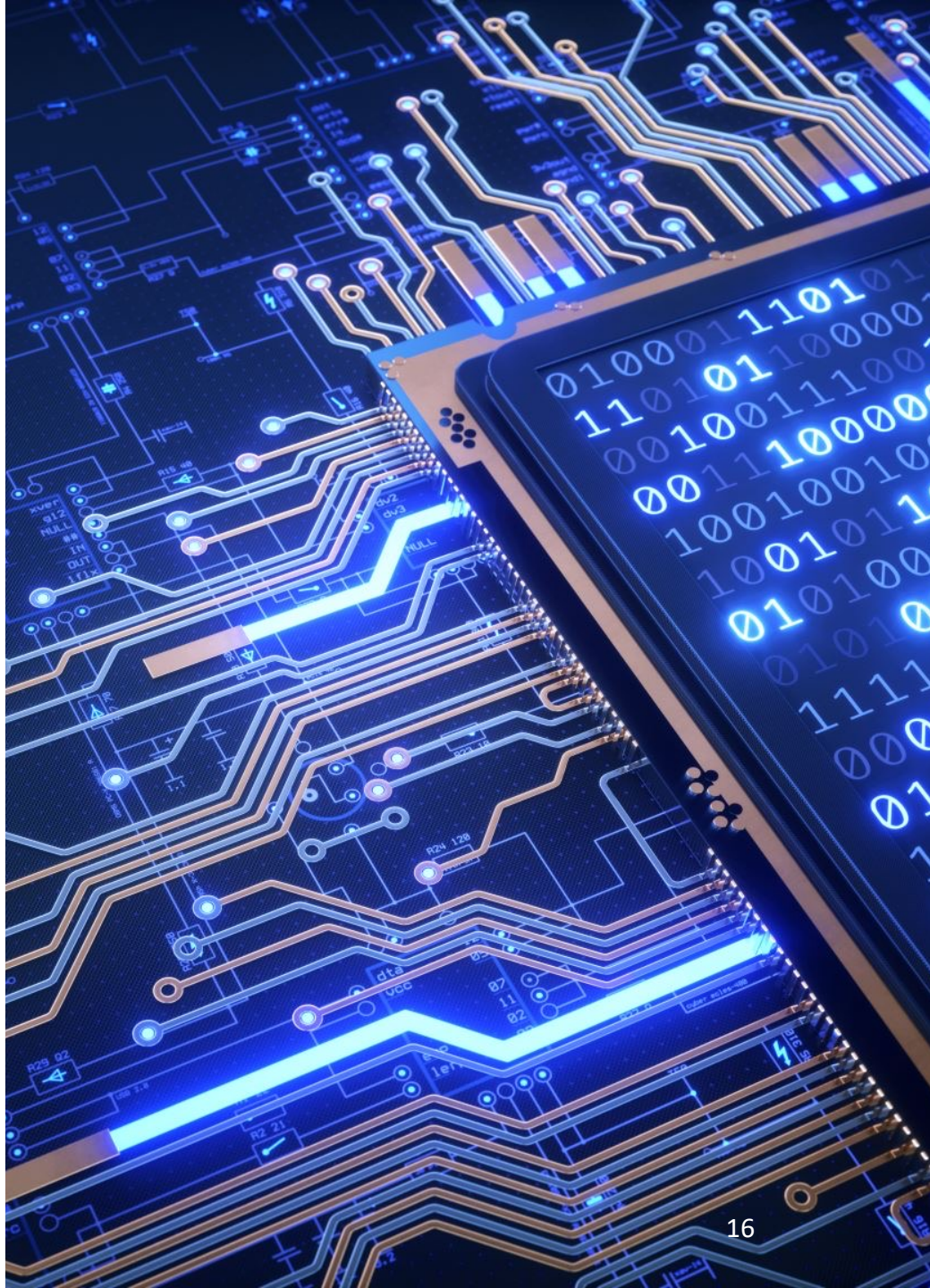


Undirected acyclic graph



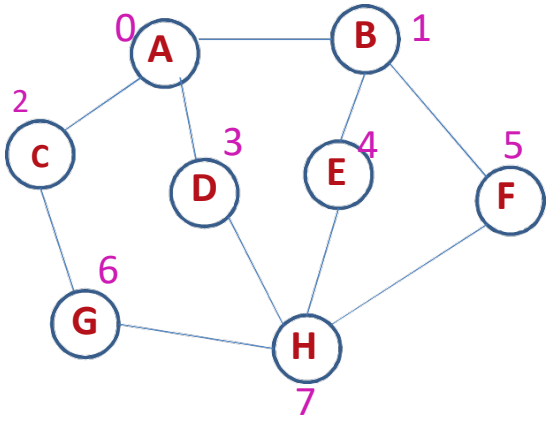
Directed acyclic graph

- Cycles cause several issues in designing graph algorithms such as finding shortest path



Implementing Graphs

- ***Adjacency Matrix***:
- One possible way of representing graphs is by using a two-dimensional array or ***matrix*** for storing edges

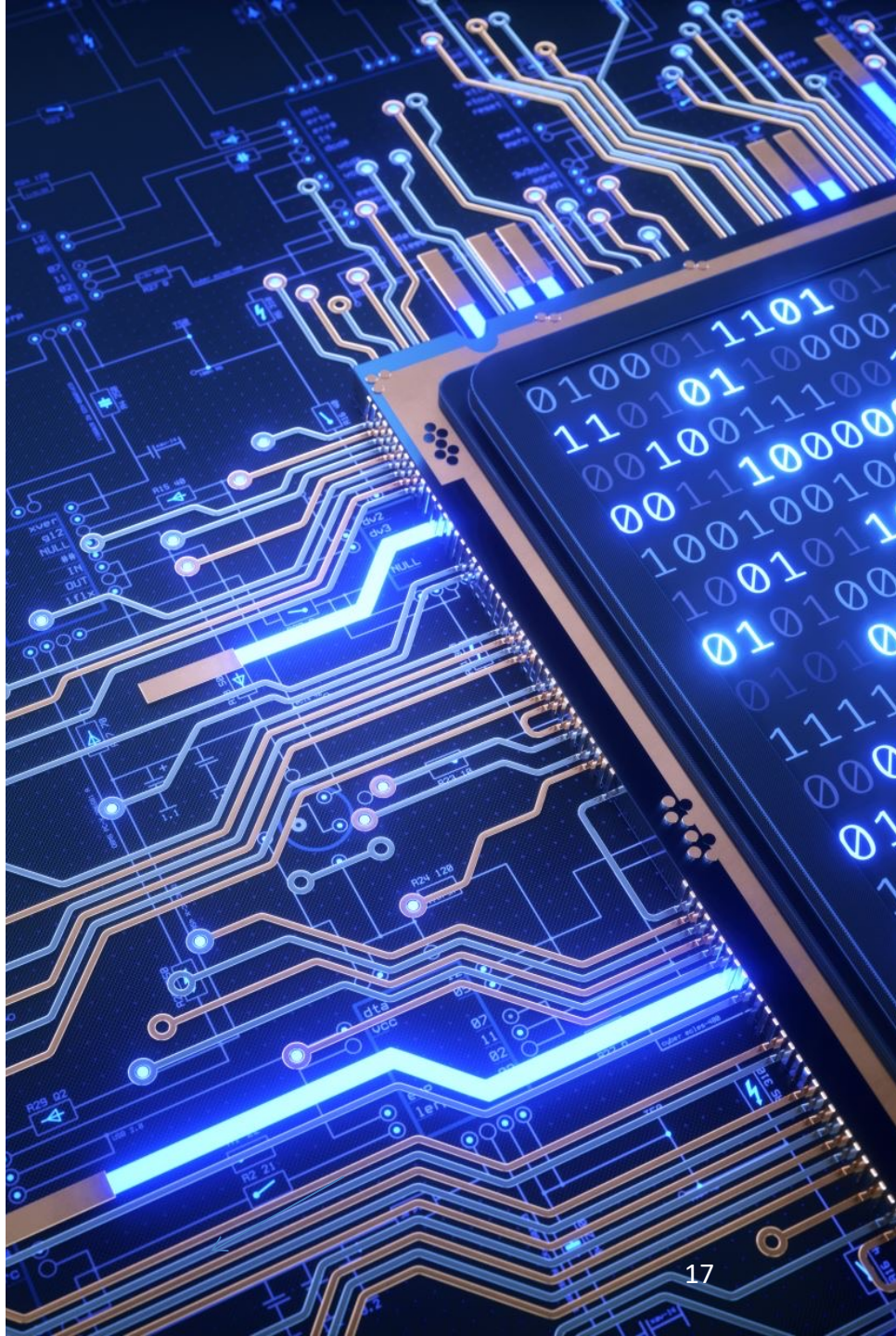


Vertex list		0	1	2	3	4	5	6	7
0	A	0	1	1	1	0	0	0	0
1	B	1	0	0	0	1	1	0	0
2	C	1	0	0	0	0	0	1	0
3	D	1	0	0	0	0	0	0	1
4	E	0	1	0	0	0	0	0	1
5	F	0	1	0	0	0	0	0	1
6	G	0	0	1	0	0	0	0	1
7	H	0	0	0	1	1	1	1	0

Adjacency matrix

Making an entry in the matrix:

$$X_{ij} = \begin{cases} 1, \text{ if } \exists \text{ edge from i to j} \\ 0, \text{ otherwise} \end{cases}$$



Implementing Graphs

- ***Adjacency Matrix***:
- For an ***undirected graph***, adjacency matrix would be *symmetric*, because $X_{ij} = X_{ji.}$ So , it will be enough to go through either upper triangle or lower triangle to see all the edges.

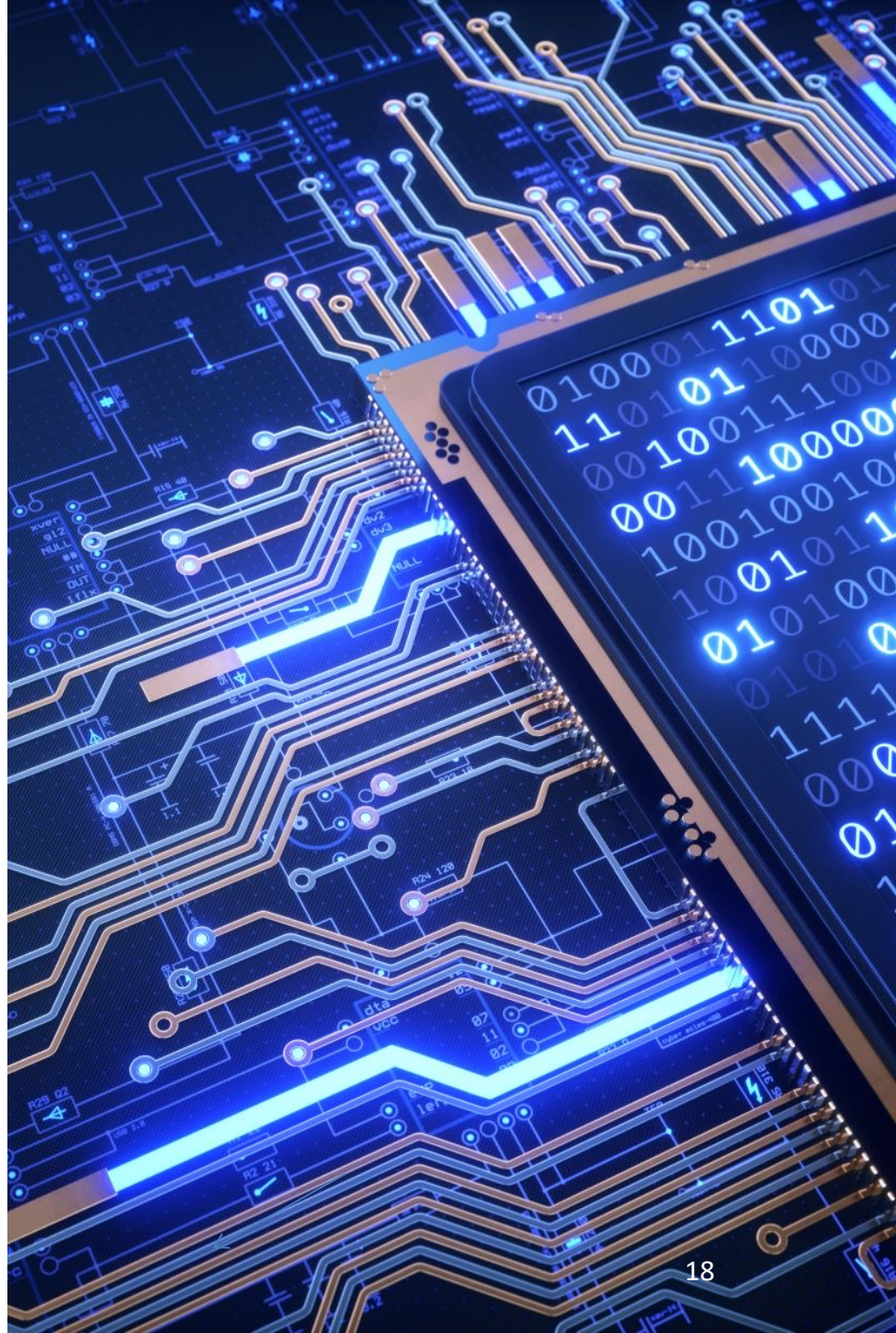
Vertex list	
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

X	0	1	2	3	4	5	6	7
0	0	1	1	1	0	0	0	0
1	1	0	0	0	1	1	0	0
2	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	0	1
6	0	0	1	0	0	0	0	1
7	0	0	0	1	1	1	1	0

Adjacency matrix

- Time complexity = O(|E|)

Since the maximum number of edges in a simple graph is of O(n^2), where n is the number of vertices, the overall time complexity is O(n^2)



Implementing Graphs

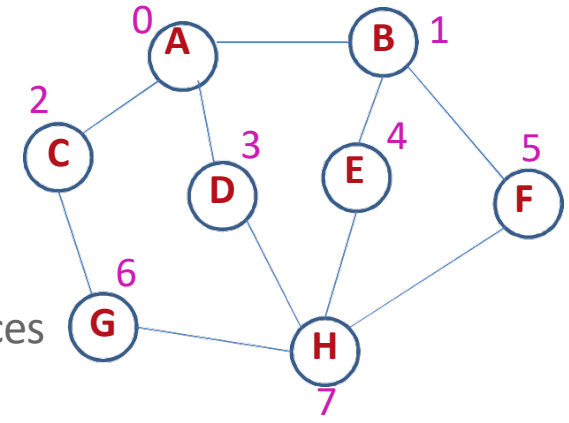
Adjacency List:

X				
0	1	2	3	
1	0	4	5	
2	0	6		
3	0	7		
4	1	7		
5	1	7		
6	2	7		
7	3	6	4	5

Adjacency lists (array representation)

In real world, $|E| < n^2$,

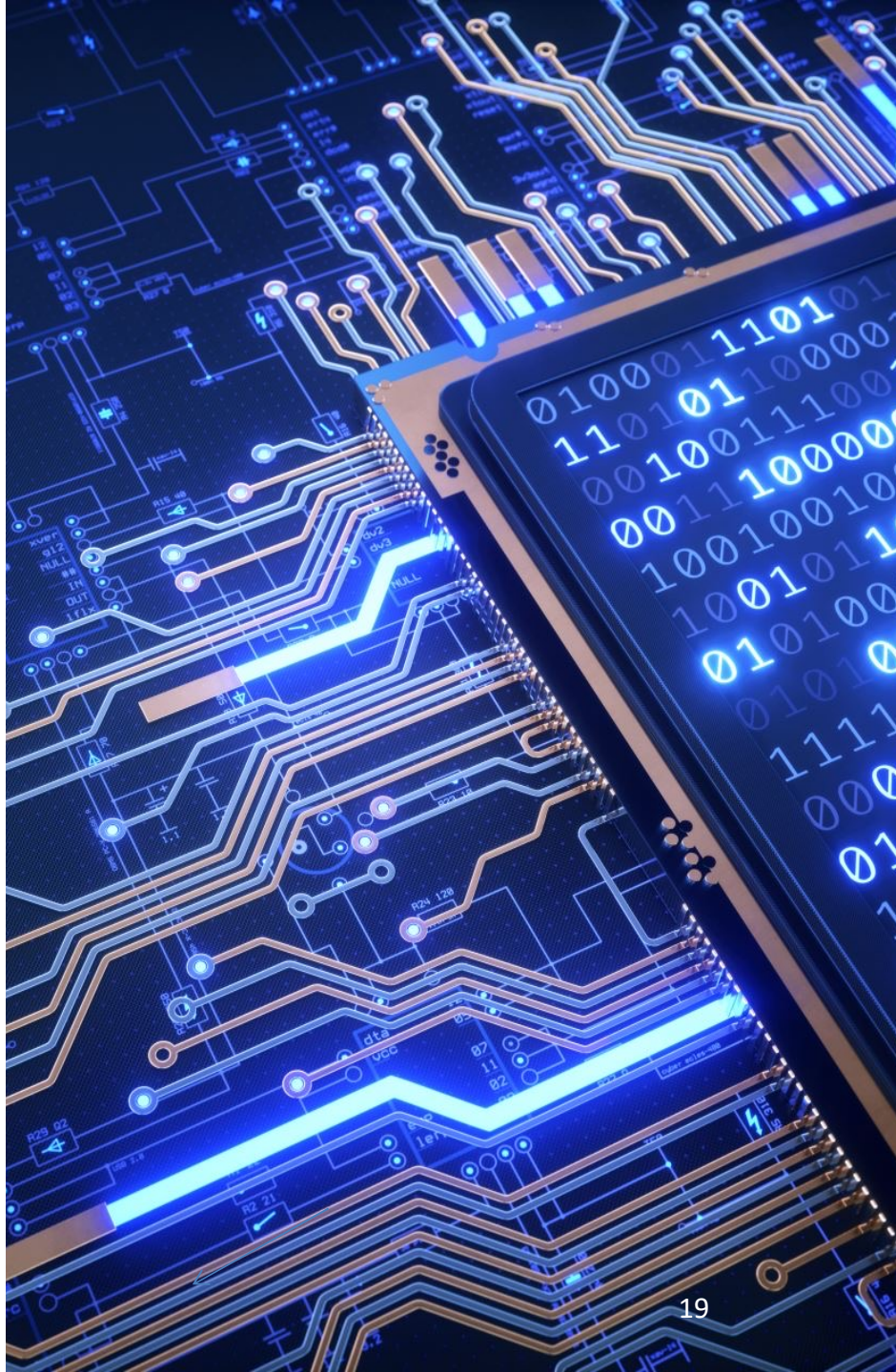
where n is the number of vertices



Adjacency list is best suitable for sparse graphs

Time complexity of finding whether two vertices are connected is **O(*n*)**, in case of linear search

Time complexity of finding all neighbor vertices is **O(*n*)**



The Graph ADT

Abstract DataType Graph

{

instances

collection of elements called nodes or vertices and edges; both edges and vertices may be associated with some values

operations

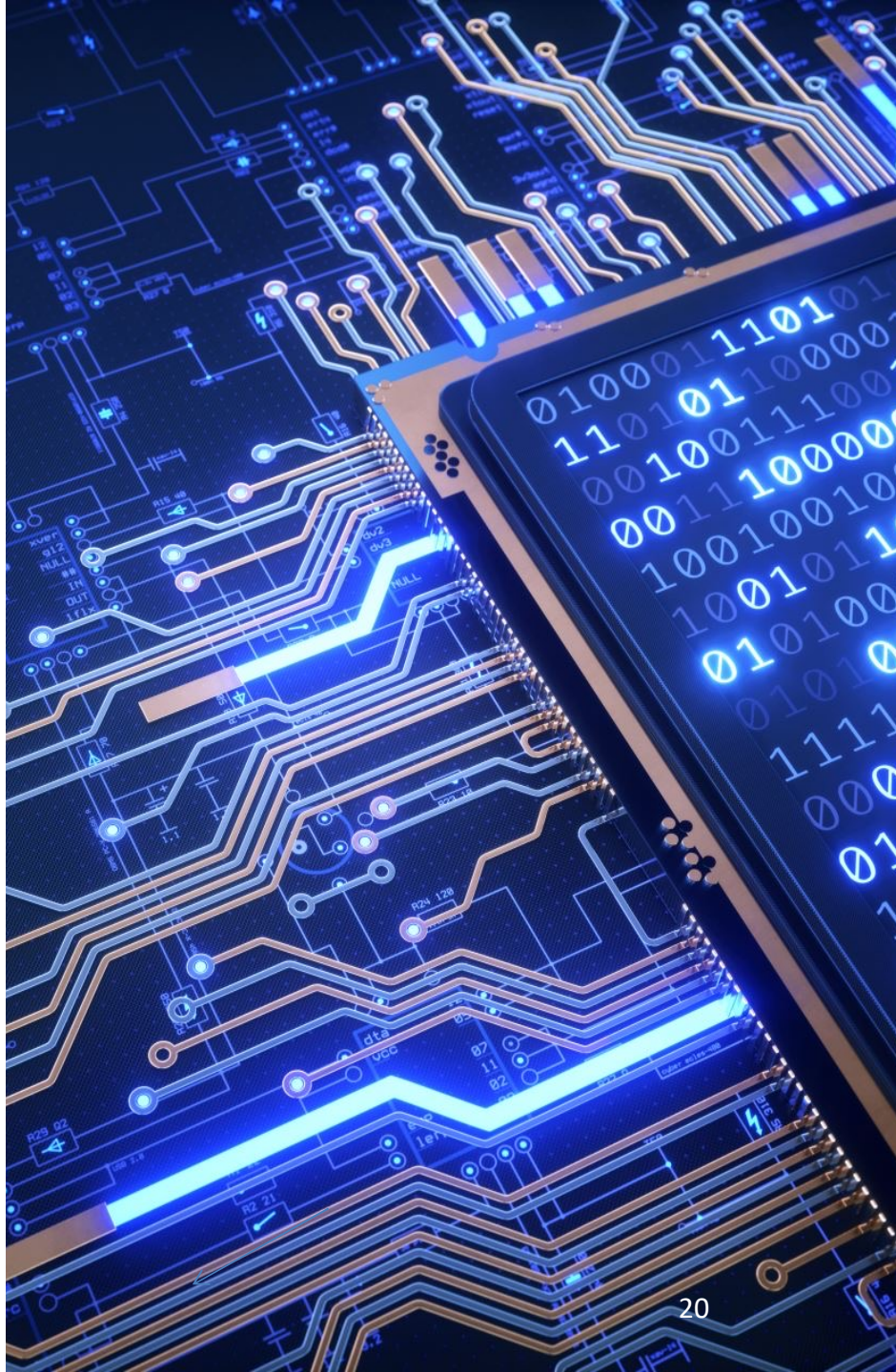
adjacent(G, *x*, *y*): tests whether there is an edge from the vertex x to y; *neighbors*(G, *x*): lists all the vertices directly connected to *x*

addVertex(G, *x*): adds the vertex *x* to the graph, if not present

removeVertex(G, *x*): removes the vertex *x*, if present

addEdge(G, *x*, *y*): adds an edge connecting the vertices *x* and *y*, if not existing

removeEdge(G, *x*, *y*): removes the edge connecting the vertices *x* and *y*, if existing

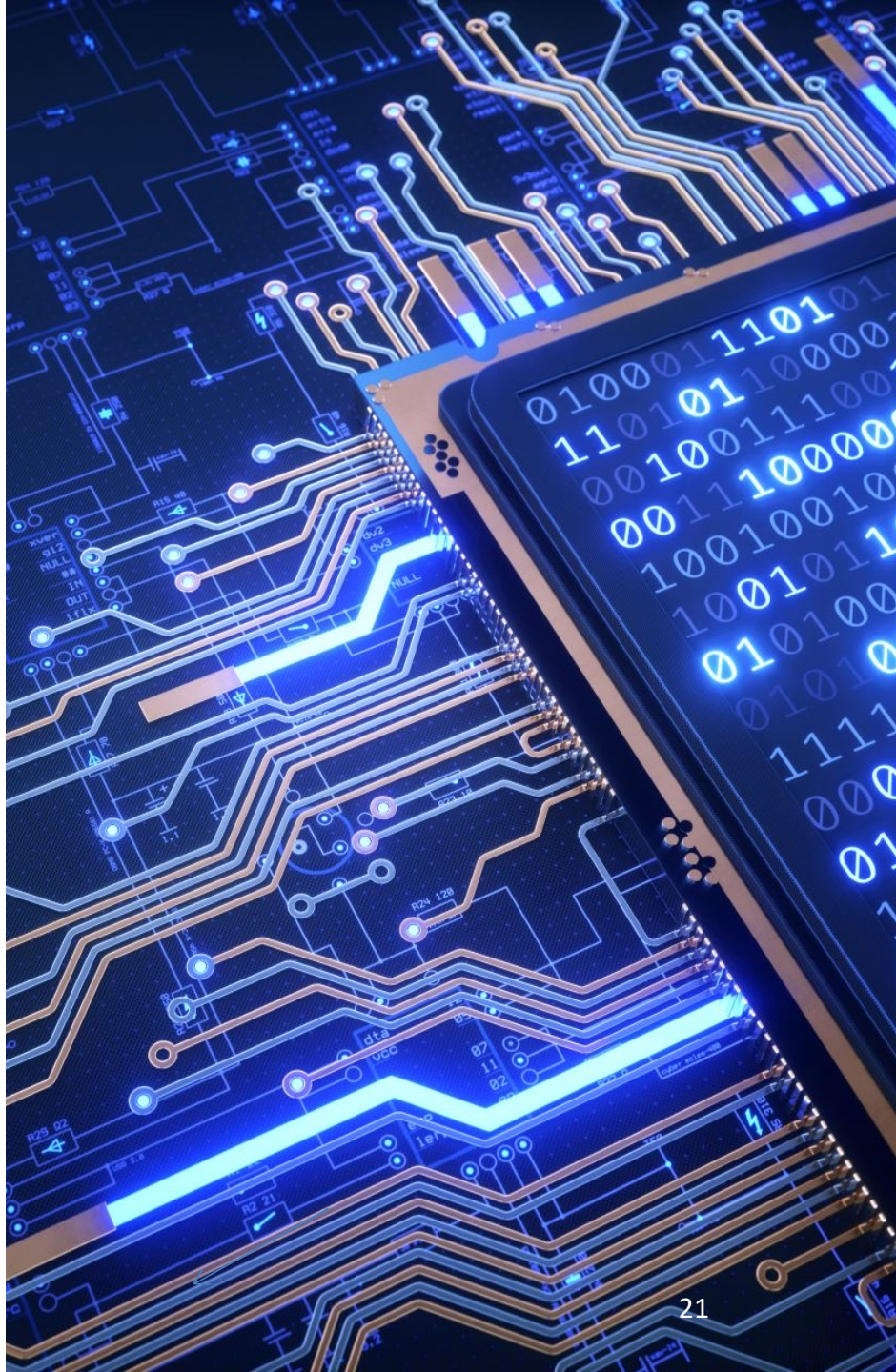


The Graph ADT

Abstract DataType Graph

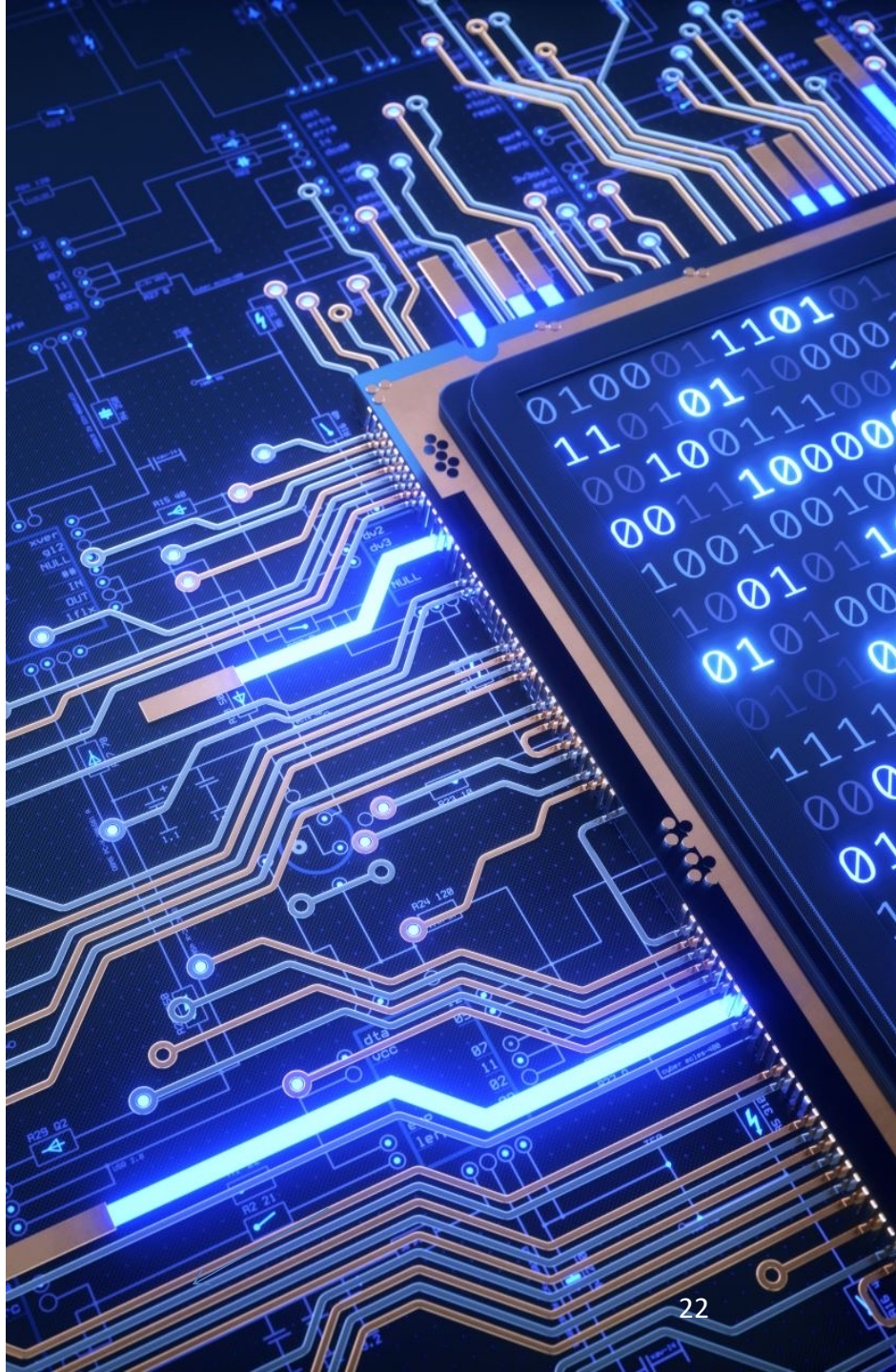
getVertexValue(G, *x*): returns the value associated with the vertex *x*
setVertexValue(G, *x*, *v*): sets the value of the vertex *x* to *v*
getEdgeValue(G, *x*, *y*): returns the value associated with the edge (*x*, *y*) *setEdgeValue*(G, *x*, *y*, *v*): sets the value of the edge (*x*, *y*) to *v*
DepthFirst(): Returns the depth first traversal list of vertices
BreadthFirst(): Returns the breadth first traversal list of vertices

}



Graph Traversal

- A fundamental problem concerning the graph data structure is the ***reachability problem***.
- In simple form, reachability problem requires us to determine whether there exists a path in the given graph G = (V, E) such that this path starts at vertex *v* and ends at vertex *u*
- A more general form is, to determine for a given starting vertex
  $v \in V$ all vertices *u* such that there is a path from *v* to *u*
- The second problem can be solved by starting at vertex *v* and systematically searching (traversing) the graph G for vertices that can be reached from v
- There are two ways to traverse a graph:
  1) Breadth First Traversal
  2) Depth First Traversal

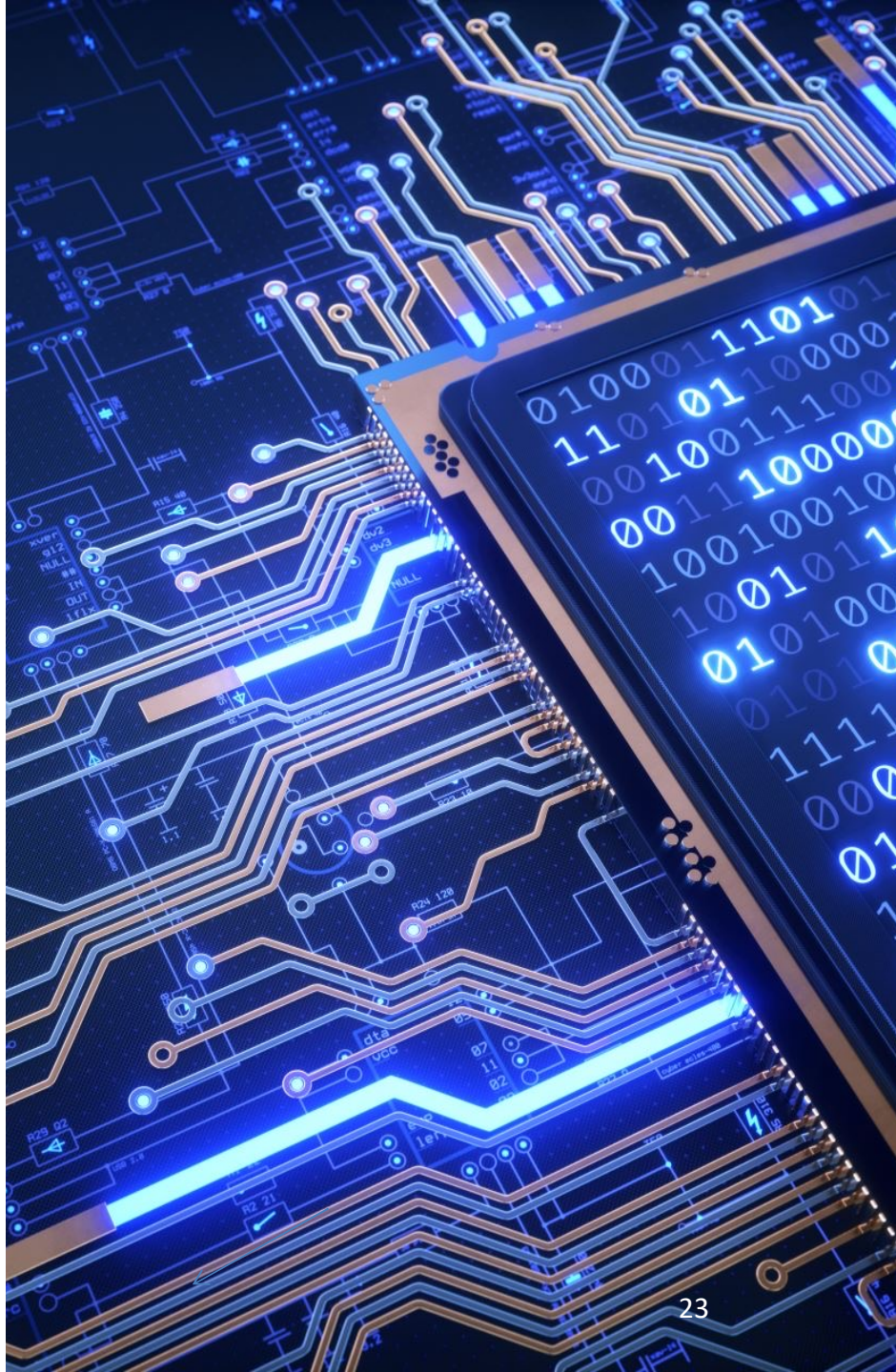


Graph Traversal

- **Breadth First Search/Traversal (BFS):**

  – Needs a ***queue*** data structure to keep track of unexplored nodes

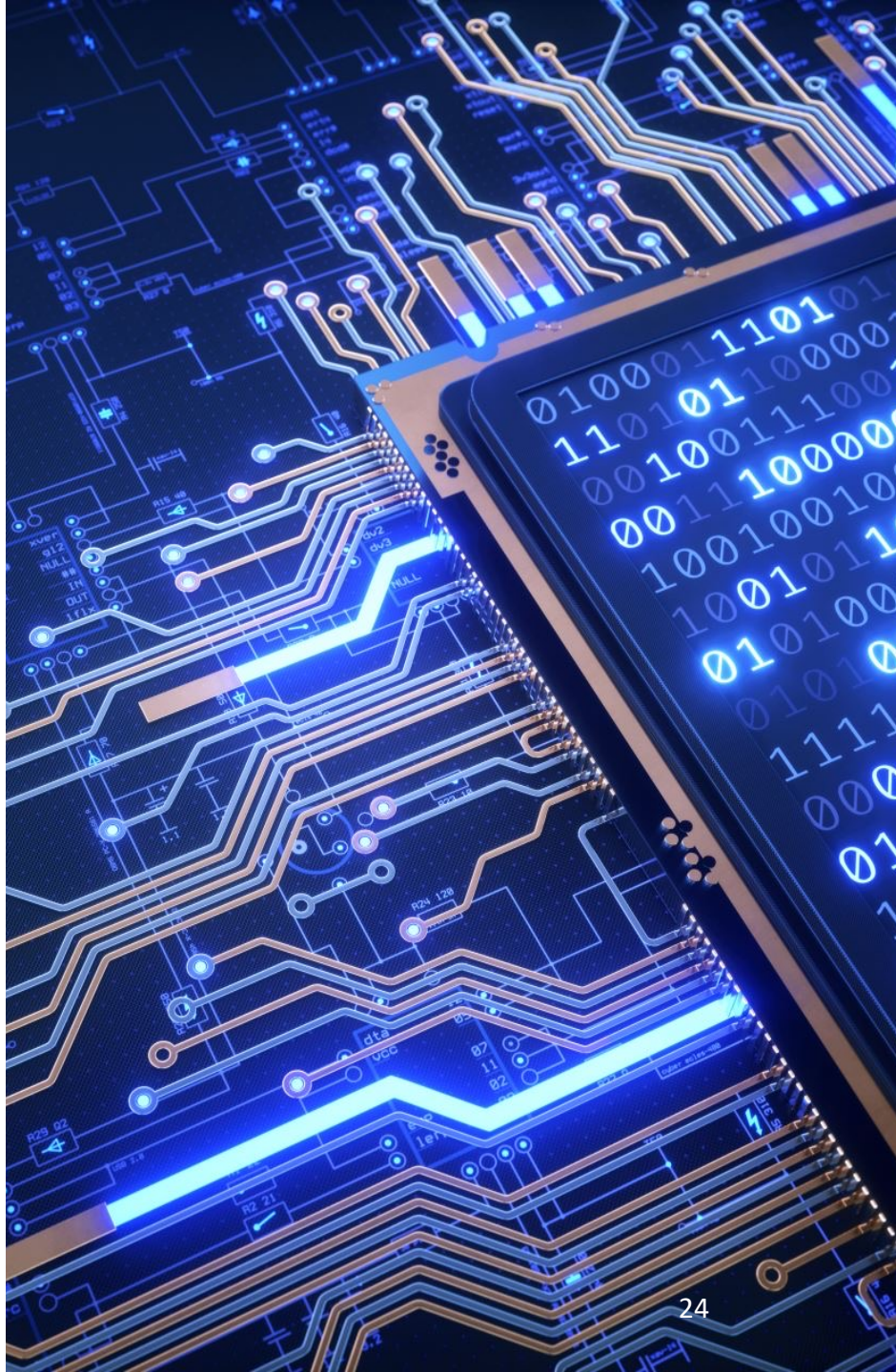
1) Start at a vertex *v* and mark it as visited
2) All unvisited vertices adjacent from *v* are visited next. These are unexplored and vertex *v* has now been explored.
3) The newly visited vertices haven't been explored and are put onto the end of a queue of unexplored vertices
4) The front vertex on the queue is the next to be explored. All unvisited vertices adjacent to it are put at the end of the queue
5) Exploration continues until no unexplored vertex is left in the queue



Graph Traversal

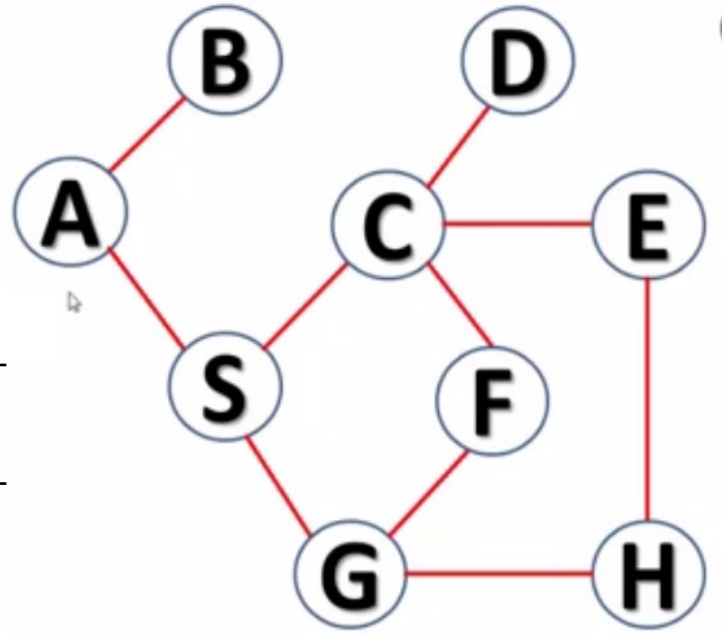
Algorithm BFS(*G*, *v*) // *v* is the start vertex of the Graph *G*

1. Enqueue(*v*, *Q*) //*Q* is a queue of unexplored vertices
2. visited[*v*] = true
3. while (*q* is not empty)
4. *u* = Dequeue(*Q*) //Mark next unexplored vertex as current
5. for (all vertices *w* adjacent from *u*)
6. if (visited[*w*] = false)
7. Enqueue(*w*, *Q*)
8. visited[*w*] = true



Graph Traversal: BFS - Example

Assume vertex A is traversed first, determine the output based on BFS

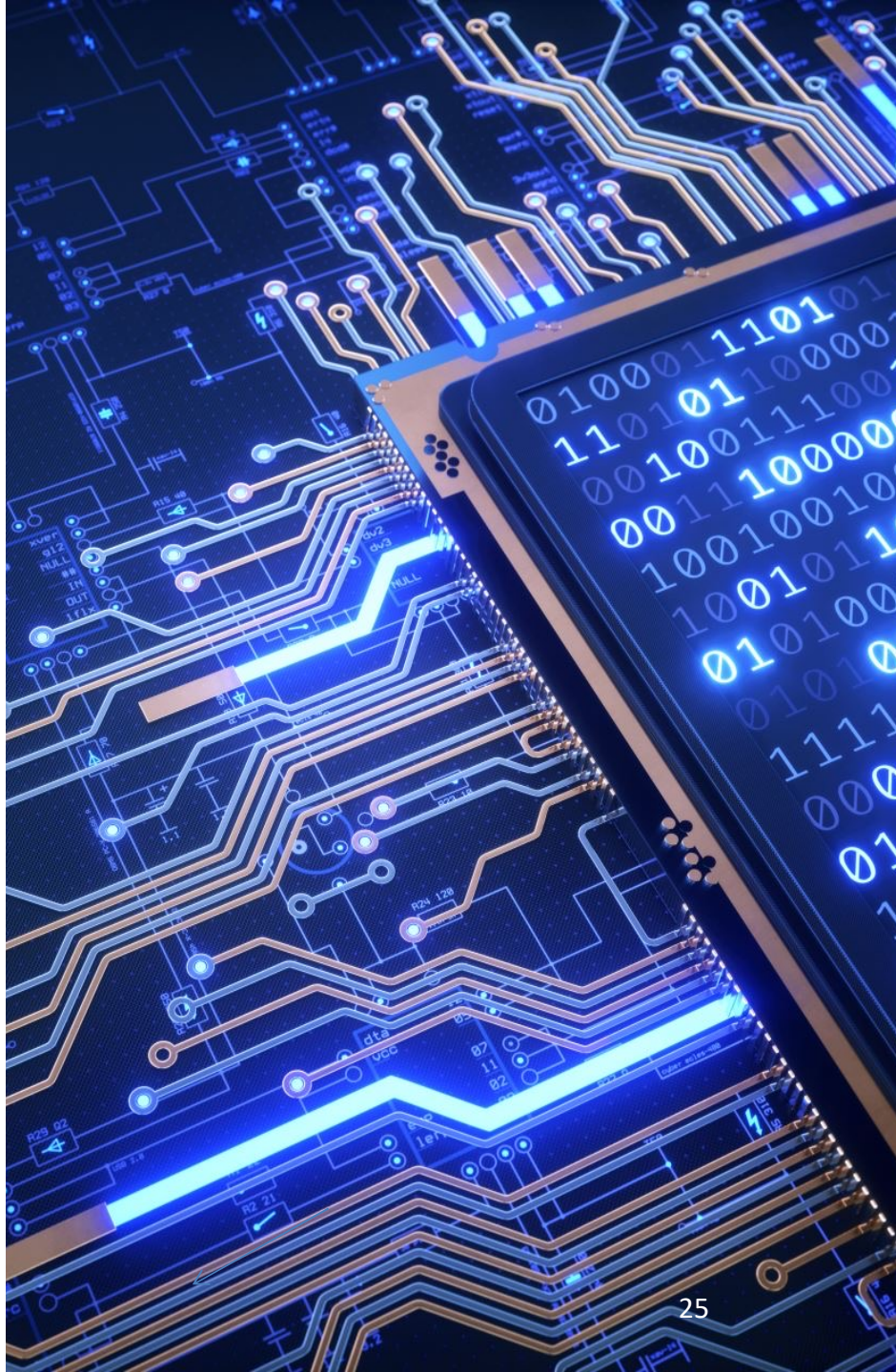


Queue:

	B	**S**		

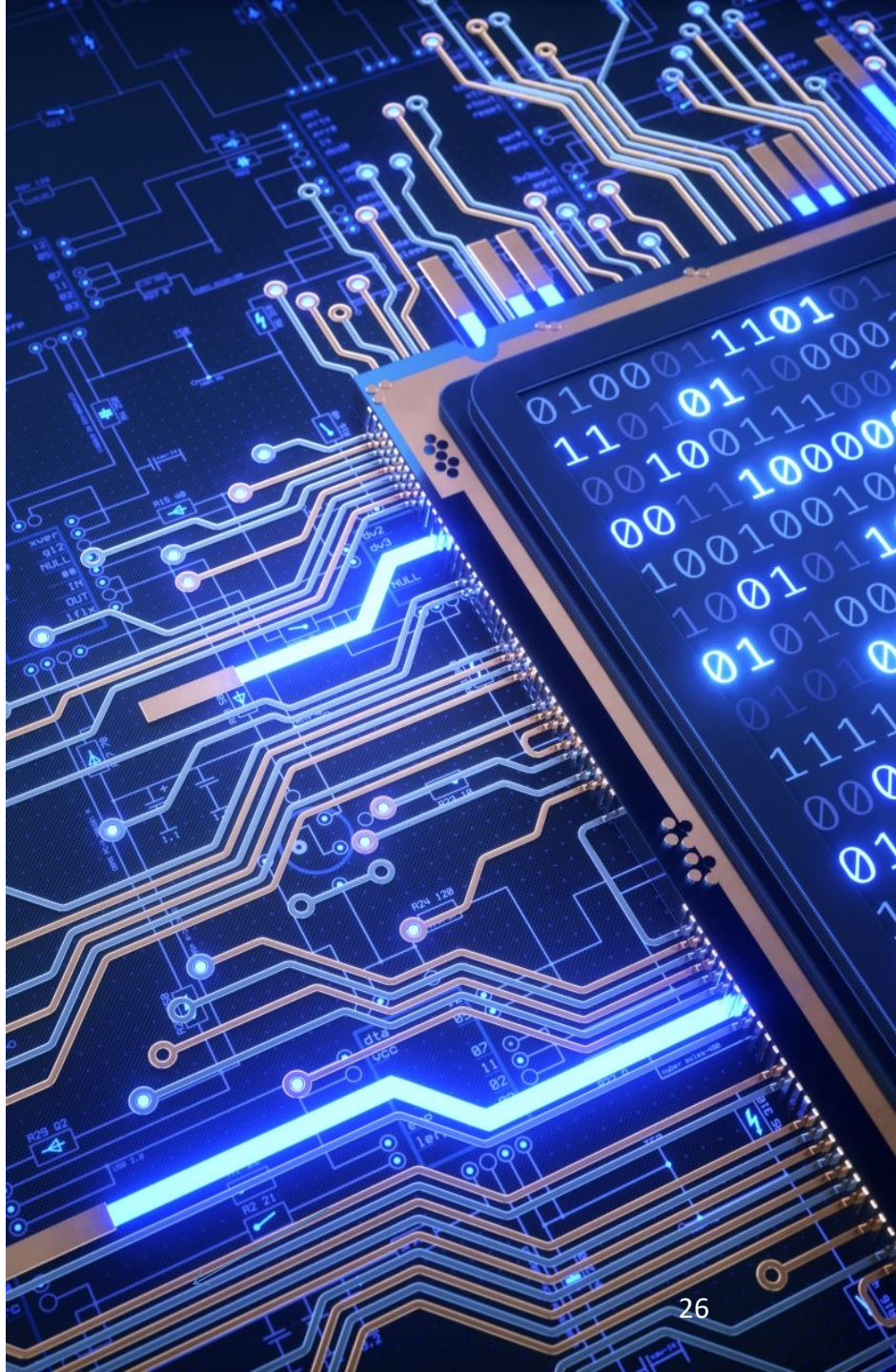
Time complexity: For a Graph *G* with *n* vertices and *e* edges
$T(n) = O(n + e)$

OUTPUT : A B S C G D E F H



Graph Traversal

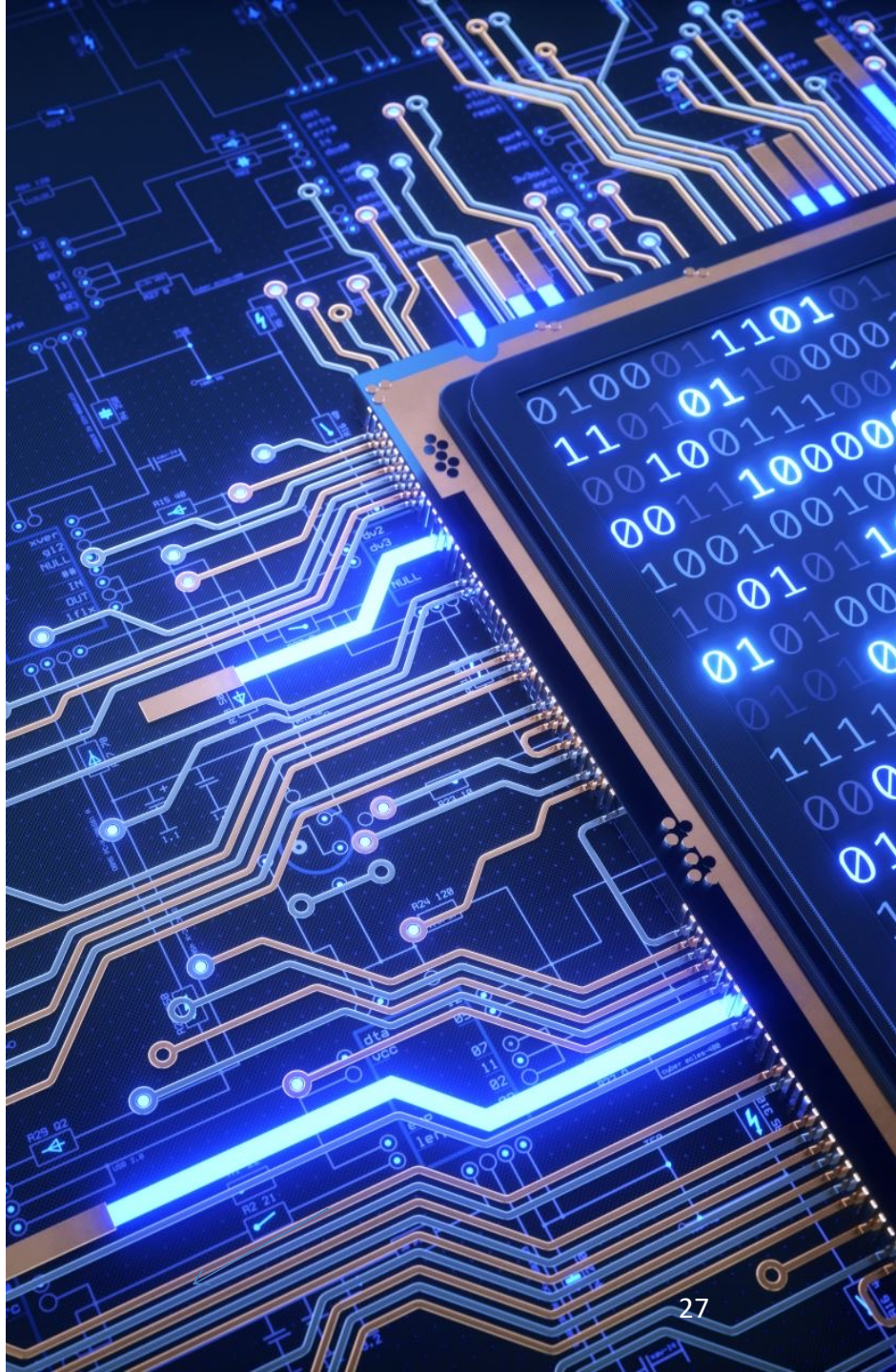
- **Depth First Search/Traversal (DFS):**
  - Needs a **stack** data structure to keep track of unexplored nodes
  - A *depth first search* (DFS) of a graph differs from a *breadth first search* (BFS) in that the exploration of a vertex *v* is suspended as soon as a new vertex *u* is reached
  - At this time, the exploration of the new vertex *u* begins. When this new vertex has been explored, we continue to explore *v*. The search terminates when all reached vertices have been fully explored.
  - Whereas, in BFS a node is fully explored before the exploration of any other node begins.



Graph Traversal: DFS Algorithm (Recursive)

Algorithm DFS(*G*, *v*) // *v* is the start vertex of the Graph *G*

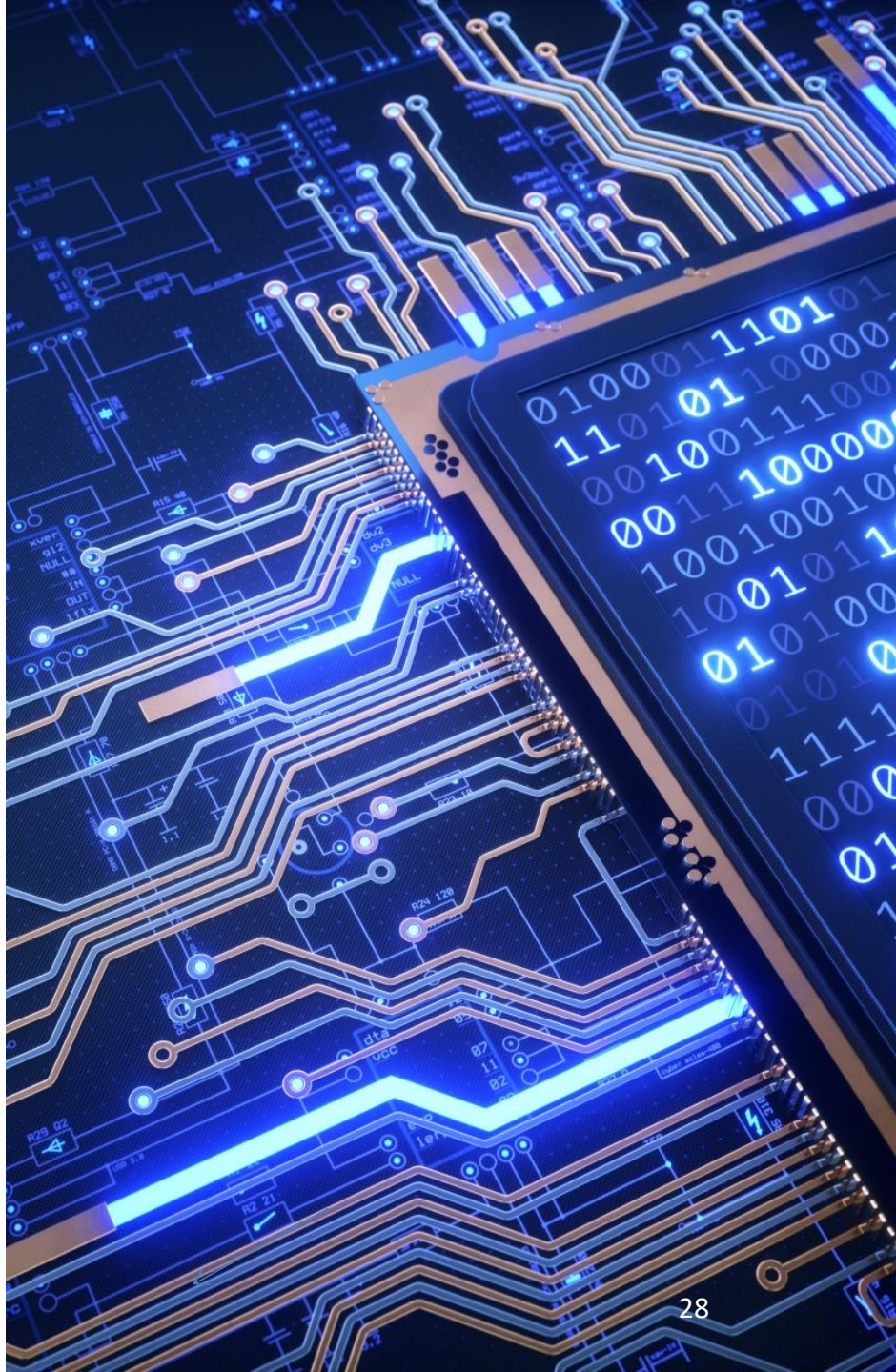
1. visited[v] = true
2. for (each vertex w adjacent from v)
3. if (visited[w] = false)
4. DFS(w)



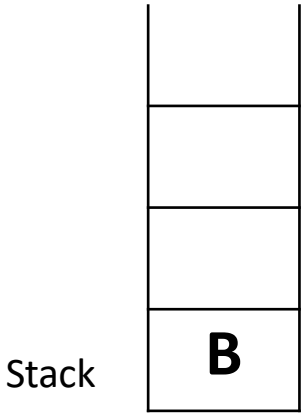
Graph Traversal: DFS Algorithm (Iterative)

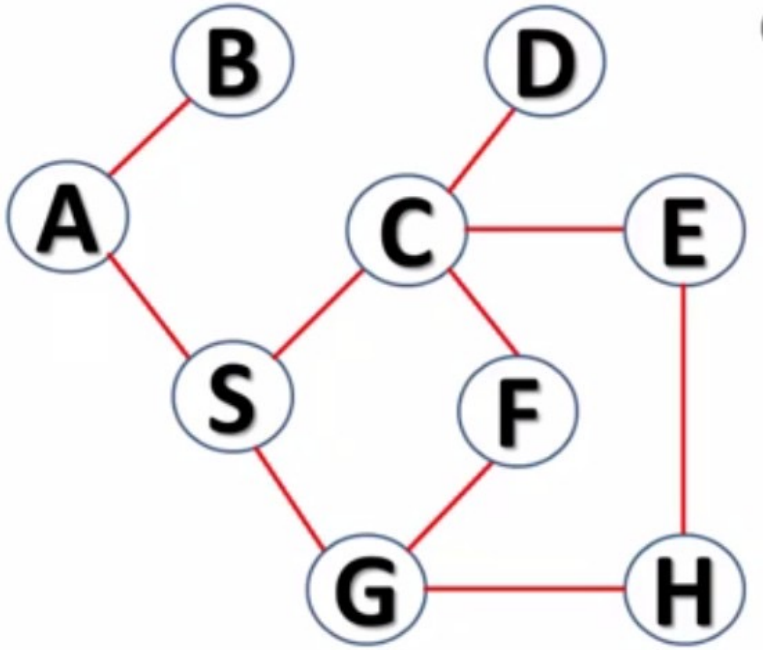
Algorithm DFS (*G*, *v*) //Where *G* is graph and *v* is the start vertex

```
Push(v, S)          //Push v on to the stack S
visited[v] = true
while(S is not empty)
    u = Top(S)      //top of the stack is the vertex to visit next
    Pop(S)
    for(all neighbors w of u in Graph G)
            if(visited[w] = false)
                    Push(w, S)
                    visited[w] = true
```



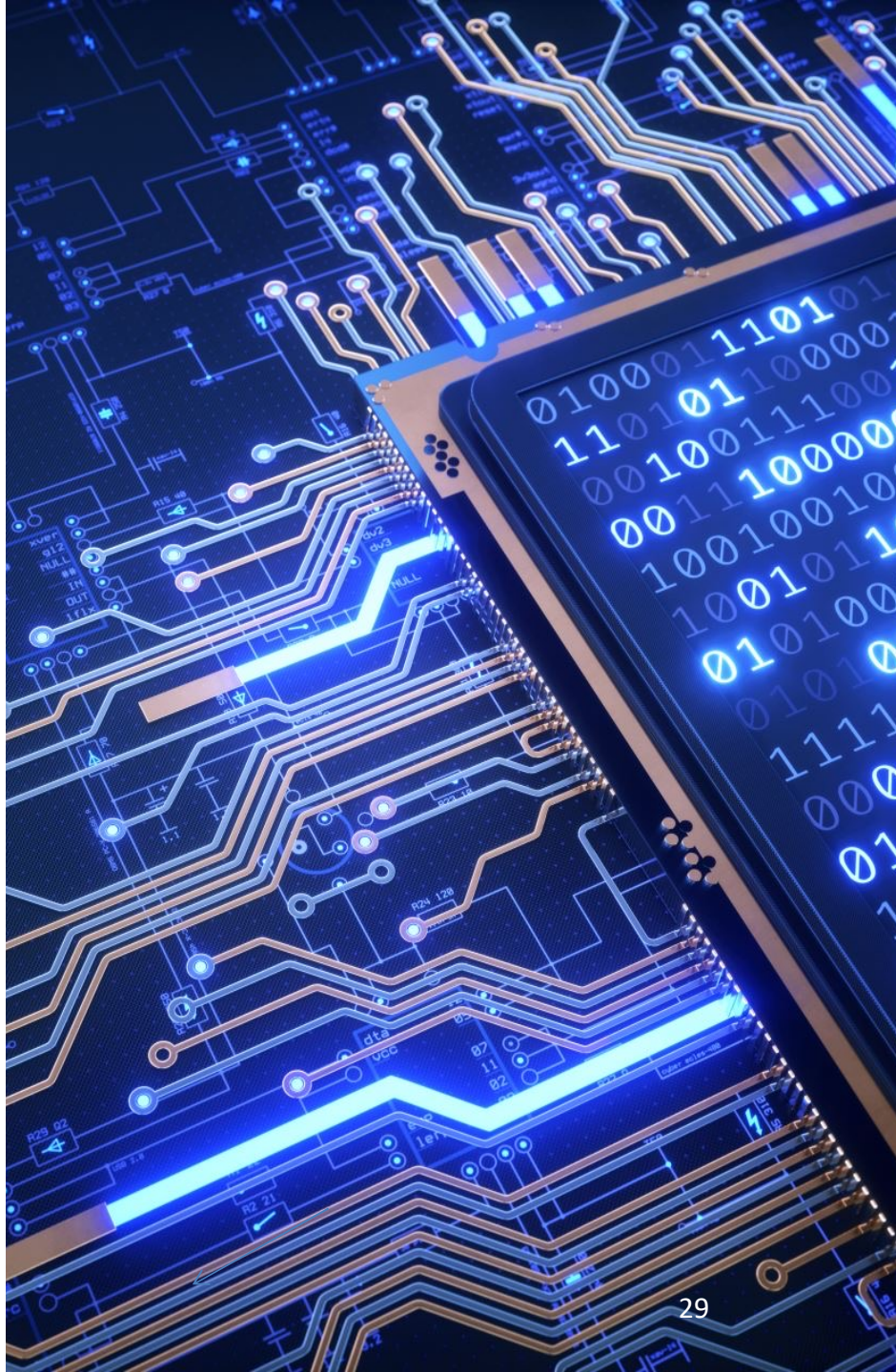
Graph Traversal: DFS - Example





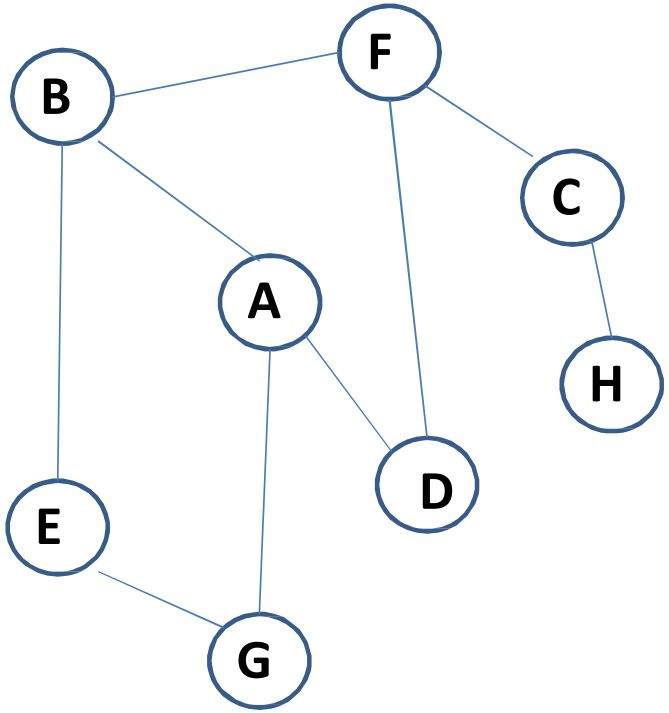
Time complexity:
For a Graph *G* with *n* vertices and *e* edges
T(*n*) = O (*n* + *e*)

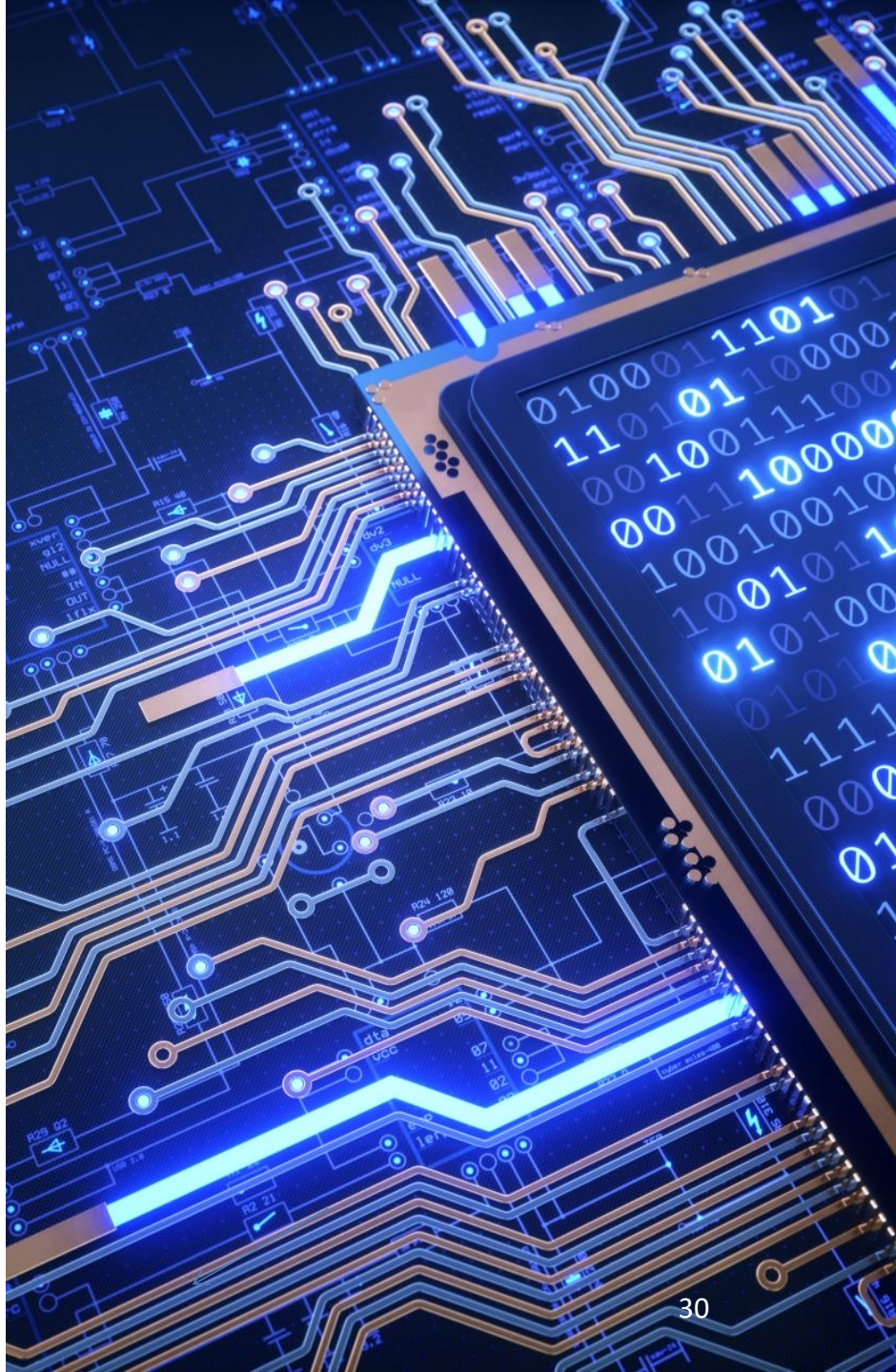
OUTPUT : A B S C D E H G F



Graph Traversal Exercise

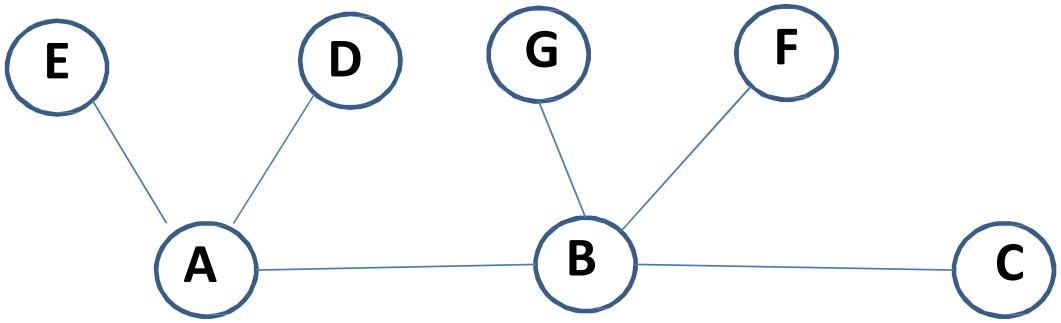
- Find the DFS and BFS outputs of the following graph.

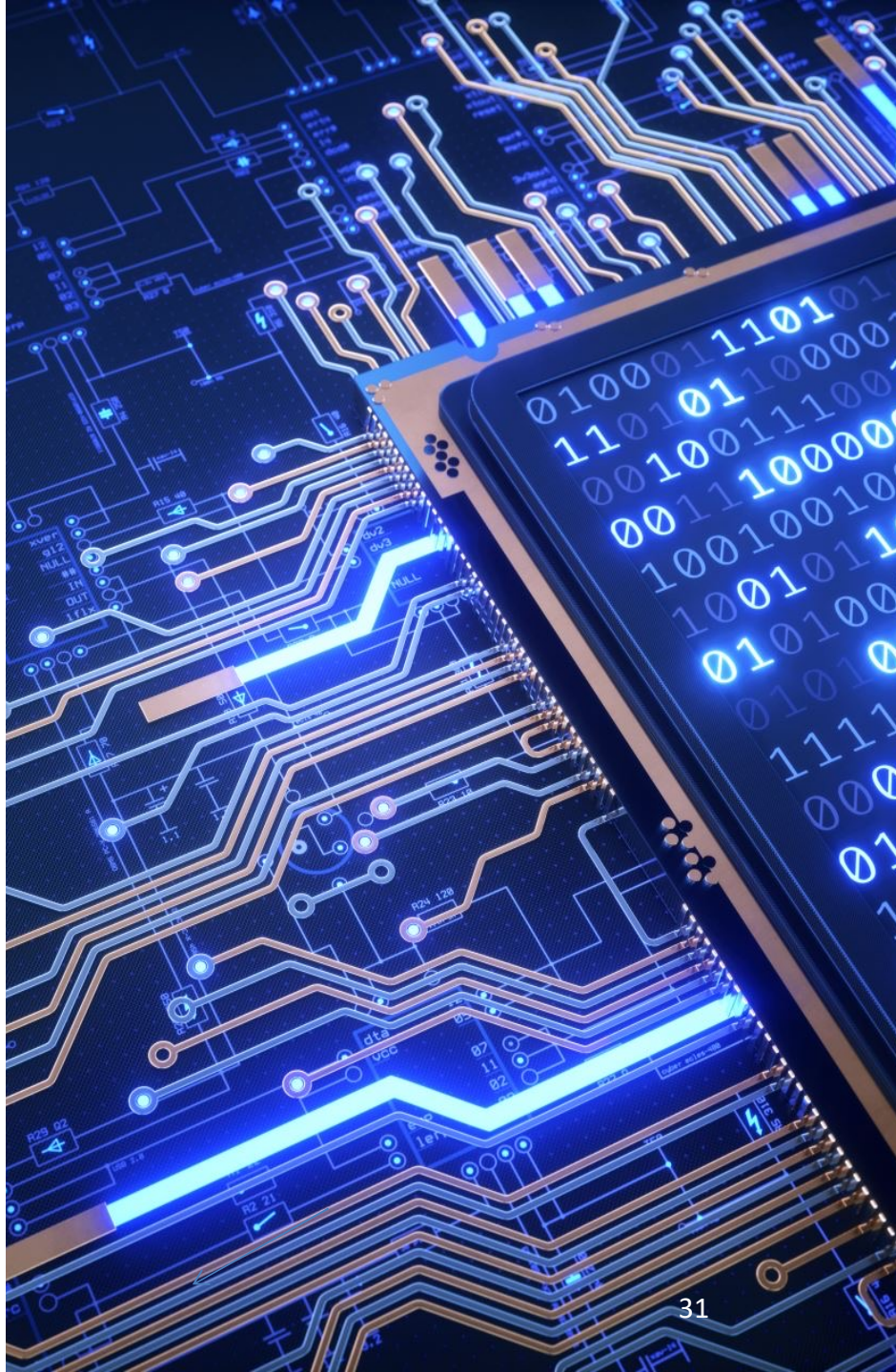




Graph Traversal Exercise

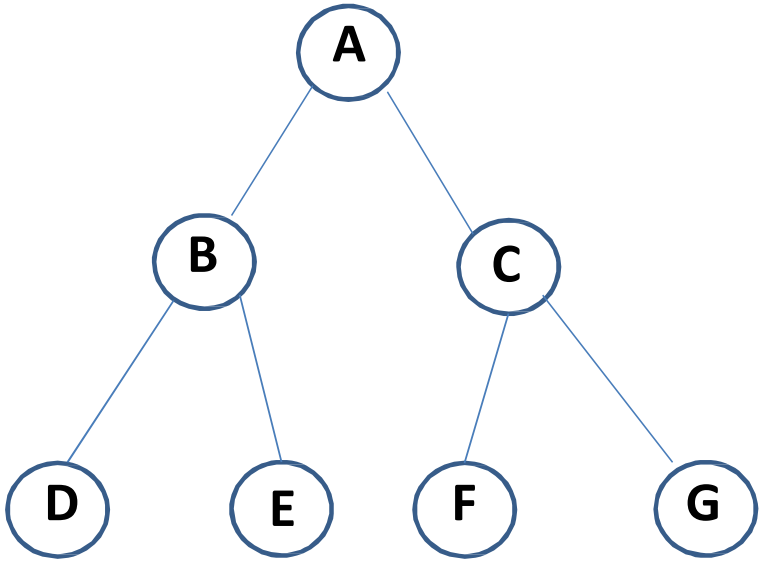
- Find the DFS and BFS outputs of the following graph.

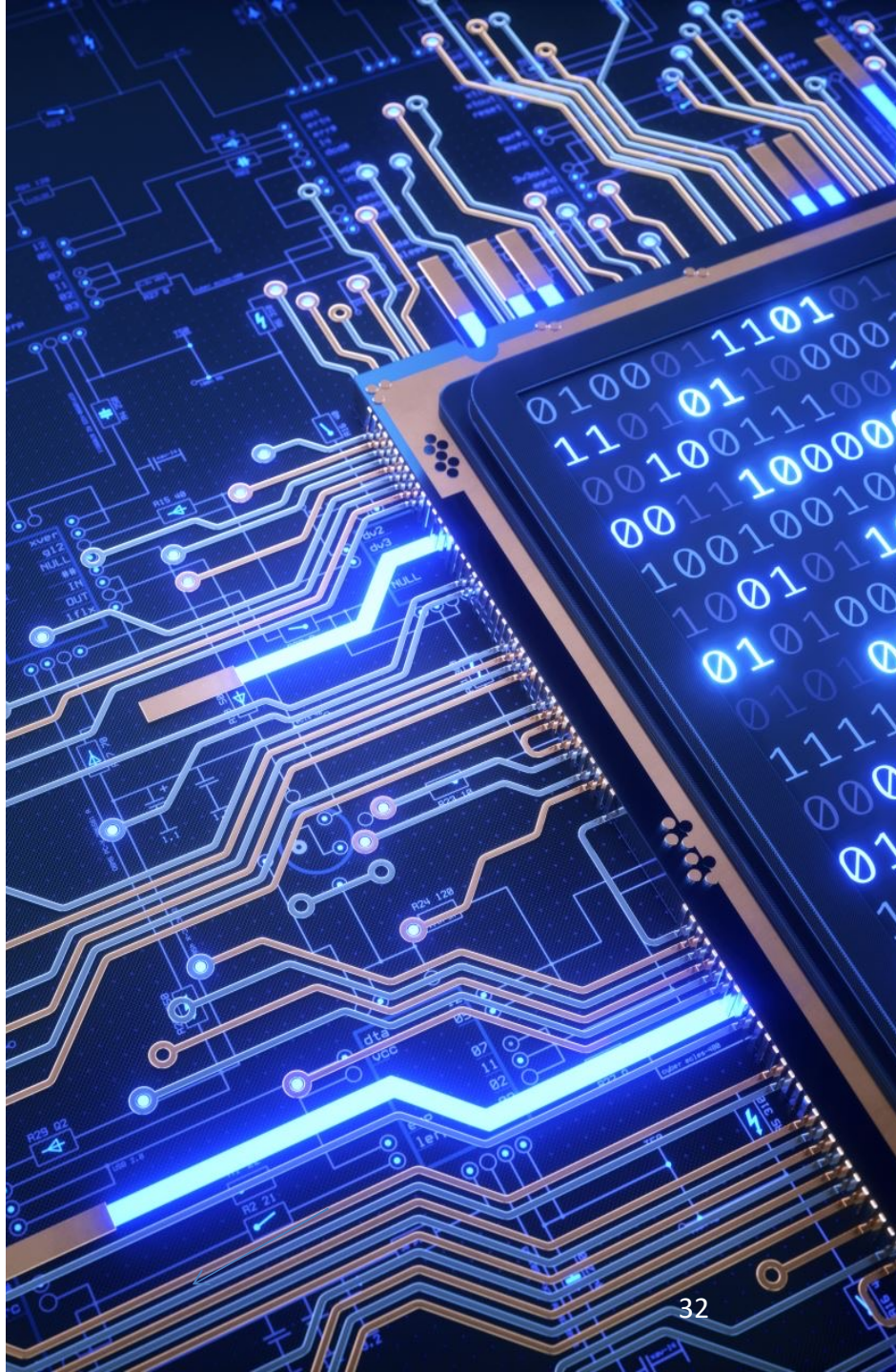




Graph Traversal Exercise

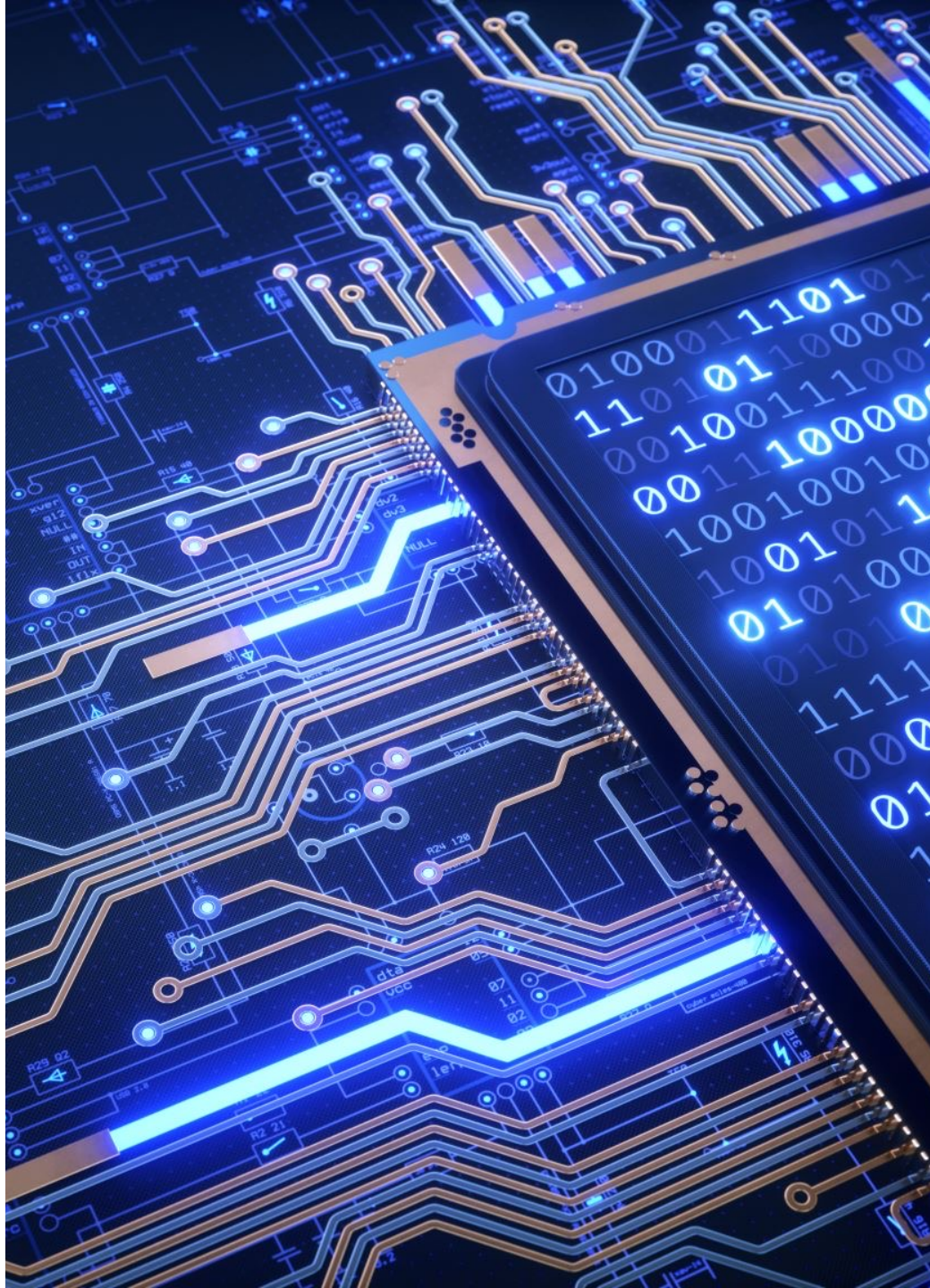
- Find the DFS and BFS outputs of the following graph.





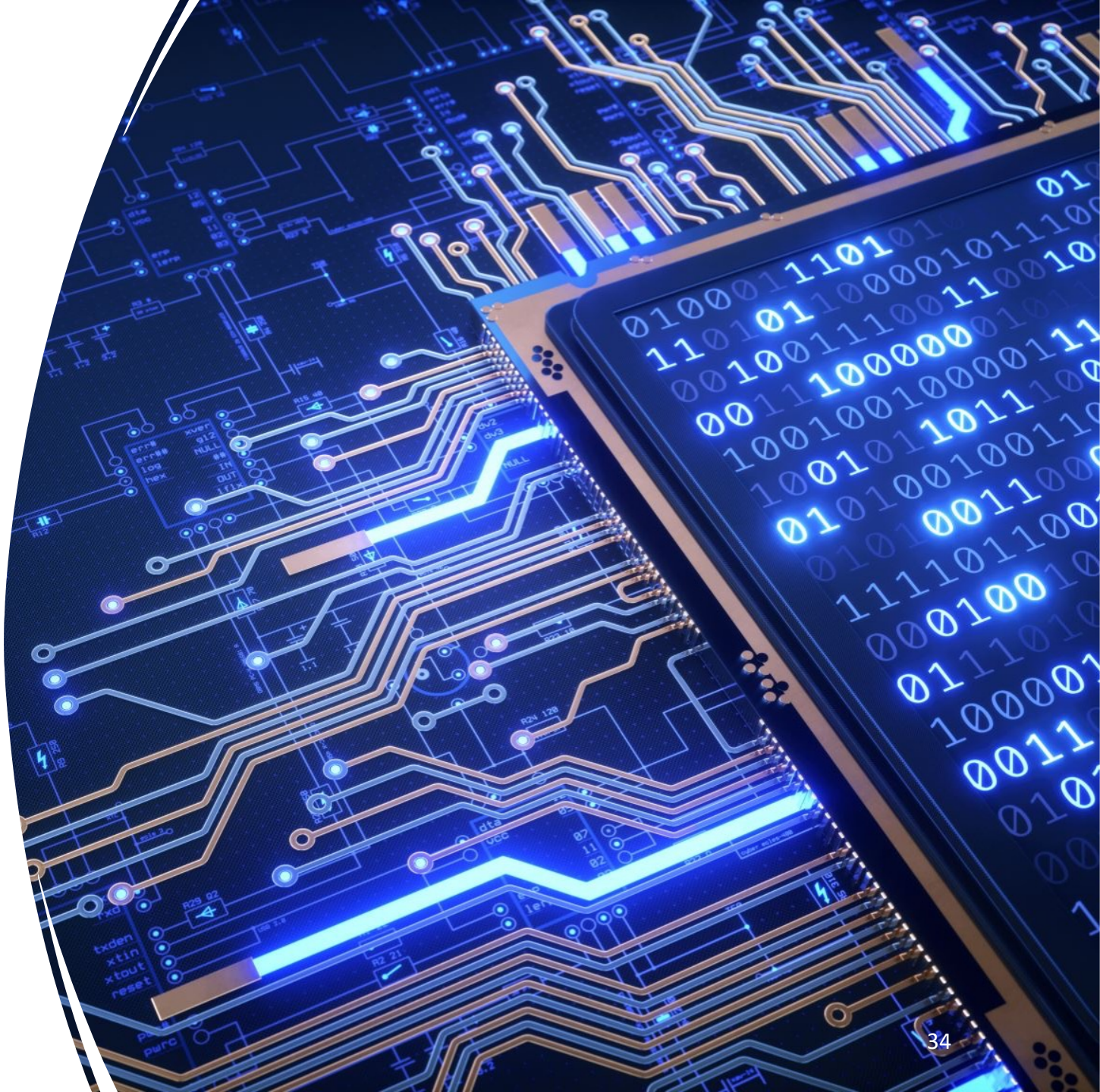
Application of Graphs

- Data structures
- Social networks
- Genomics
- Ecology
- Routing
- Supply chain management
- Molecular structure
- Market networks



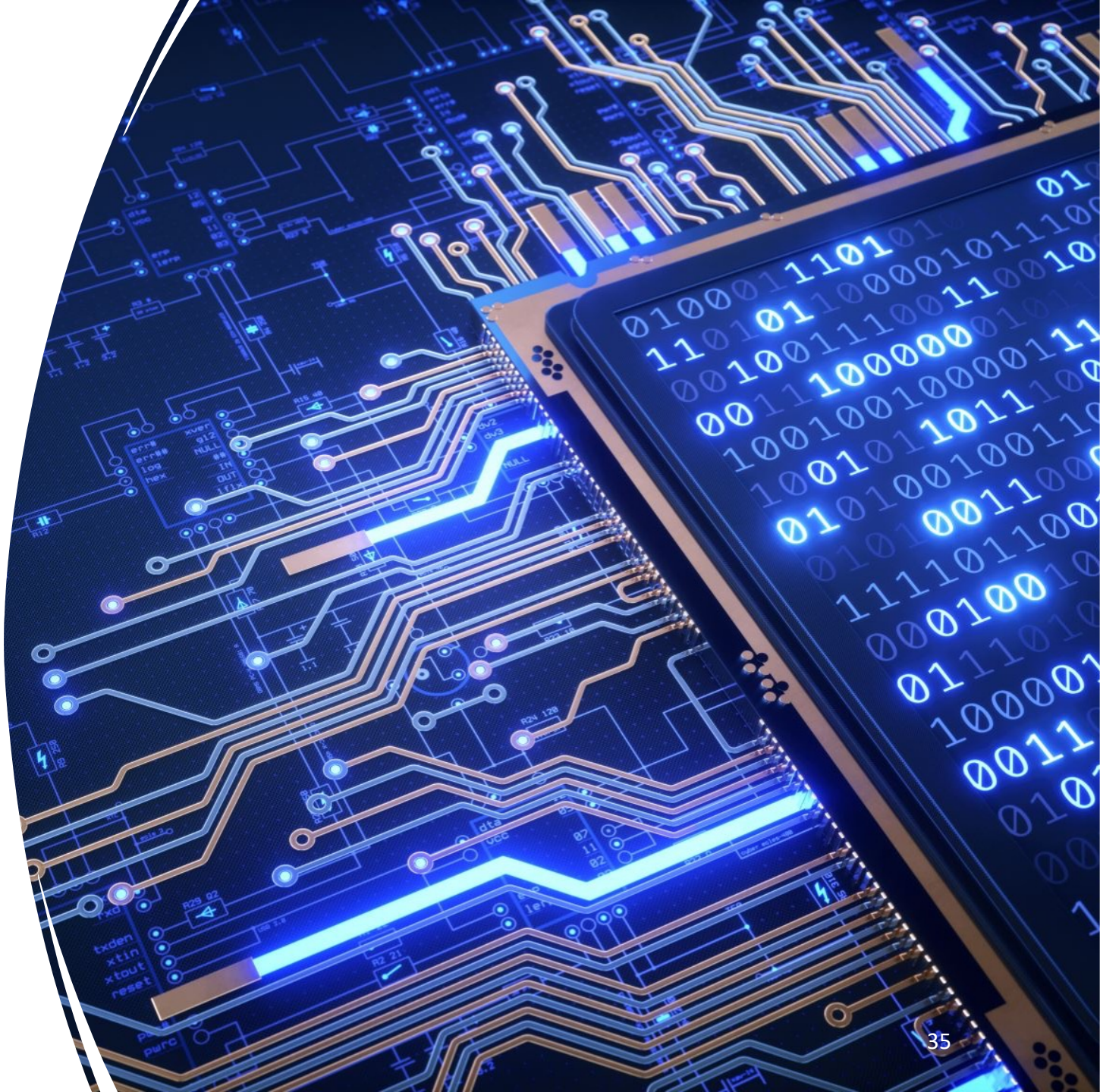
Summary

- Graphs consist of nodes (or vertices) connected by edges (or links).
- Types of Graphs
  - Directed and Undirected
  - Weighted and Unweighted
- Properties of Graphs
- Graph Traversal
  - BFS
  - DFS
- Applications



Reference

Rosen, K. H. (2012). *Discrete mathematics and its applications (7th Edition)*. McGraw-Hill.
Chapter 10



See you next time!