From Induction to Graph Theory

Motivation, Algroithms, Applications, and Correctness

by Eric Yang Xingyu

This is a handout for the lecture on graph theory for my fellows. It covers the basic concepts of graph theory, including the definition of graphs, trees, and the basic algorithms for graph traversal. The handout also includes the correctness proof of the algorithms. Before delve into the graph theory, we will first introduce the concept of structural induction, which is the foundation of the correctness proof of the graph algorithms, and other structural algorithms.

Contents

§1. From Induction to General Induction

Remark 1.1 .
This lecture note is compiled using typst, and has two versions, one for the lecturer and one for the student. The latter version will be used during the lecture as handouts for the students. After the lecture, the lecturer version will be shared with students for further study or reference, which contains hints for lecture and solution/proofs for examples and exercises.

§1.1. Recap on Induction

Definition 1.1.1 (Induction).
Induction is a method of proof in which we prove that a statement is true for all natural numbers by proving that it is true for the smallest natural number and then proving that if it is true for some natural number, then it is true for the next natural number.

The most common use case of induction is to prove a statement defined on the natural numbers.

Example 1.1.1 .
Prove that $1 + 2 + 3 ... + n = \frac{n(n+1)}{2}$ for all $n \geq 1$.

Proof. Not discussed as it is trivial. □

§1.1.1. Motivation & Assumption

The principle of induction is based on the completeness of the natural numbers, which is the foundation of the induction. The completeness of the natural numbers is the property that every non-empty subset of the natural numbers has a least element. This property is the foundation of the induction, which allows us to prove a statement for all natural numbers by proving it for the smallest natural number and then proving that if it is true for some natural number, then it is true for the next natural number.

Definition 1.1.1.1 (Naive Definition of Natural Numbers).
The natural numbers are the set of positive integers, possibly including zero.

Is $0 \in \mathbb{N}$ is a question that has been debated for centuries until now. In different branches of mathematics, the definition of natural numbers varies. In this lecture, we will use $\mathbb{N}_0$ for the set of natural numbers including zero, and $\mathbb{N}$ for the set of natural numbers excluding zero.

§1.1.1.1. Peano Axioms

Axiom 1.1.1.1.1 (Peano Axioms).
The Peano axioms are a set of axioms for the natural numbers presented by the 19th-century Italian mathematician Giuseppe Peano. These axioms define the natural numbers as a set of objects, a number system, and a set of operations.

1. $0 \in \mathbb{N}$
2. $\forall n \in \mathbb{N}, n' \in \mathbb{N}$, where n' is called the successor of n
3. $\forall n \in \mathbb{N}, n' \neq 0$
4. $\forall n, m \in \mathbb{N}, n' = m' \Rightarrow n = m$
5. $\forall K \subseteq \mathbb{N}, [0 \in K \wedge \forall n(n \in K \Rightarrow n' \in K) \Rightarrow K = \mathbb{N}]$

Remark 1.1.1.1.1.
It is quite normal in both computer science and mathematics that a extremely complex system can be built on a few simple rules, or we say axioms here. Some examples are systems of differential equations, recurrence relations, etc.

Problem 1.1.1.1.1 (Some Counter Examples).

- For a number system consists of only $\{0\}$, what axioms are violated?
- For a number system consists of only $\{0, 1\}$, what axioms are violated?
- For a number system consists of only $\{0, 1, 2\}$, what axioms are violated?
- How can we find a counter example to show that axiom 5 is necessary?

§1.1.1.2. Define Arithmetic Operations on Natural Numbers

We can define the arithmetic operations on natural numbers using the Peano axioms.

Notation 1.1.1.2.1 (pred and succ).
pred() and succ() are the predecessor and successor functions on natural numbers, respectively. formally, $\text{pred} : \mathbb{N} \to \mathbb{N}$ and $\text{succ} : \mathbb{N} \to \mathbb{N}$. We will use $\text{pred}(n)$ and $\text{succ}(n)$ to denote the predecessor and successor of n in the following.

Definition 1.1.1.2.1 (Addition on Natural Numbers).
The addition of two natural numbers m and n is defined recursively as follows:

1. $m + 0 = m$
2. $m + \text{succ}(n) = \text{succ}(m + n)$

Definition 1.1.1.2.2 (Multiplication on Natural Numbers).
The multiplication of two natural numbers m and n is defined recursively as follows:

1. $m \times 0 = 0$
2. $m \times \text{succ}(n) = m \times n + m$

Problem 1.1.1.2.1 .
How can we define the **subtraction** and **integer division** on natural numbers using the Peano axioms?

§1.1.2. Weak Induction

Definition 1.1.2.1 (Weak Induction) **.**
Weak induction is a method of proof in which we prove that a statement is true for all natural numbers by proving that it is true for the smallest natural number and then proving that if it is true for some natural number, then it is true for the next natural number. Formally, let $P(n)$ be a statement for each $n \in \mathbb{N}_0$. If $P(0)$ is true, and $\forall k \in \mathbb{N}_0, P(k) \Rightarrow P(k+1)$, then $\forall n \in \mathbb{N}_0, P(n)$ is true.

Example 1.1.2.1 .
Prove that $\forall n \in \mathbb{N}_0, 3 \mid n^3 + 2n$.

Remark 1.1.2.1 .
The statement $3 \mid n^3 + 2n$ means that $n^3 + 2n$ is divisible by 3, which means that

$$\exists m \in \mathbb{Z} : n^3 + 2n = 3m.$$

Accordingly, if $3 \nmid n^3 + 2n$, then $n^3 + 2n$ is not divisible by 3, which means that

$$\exists m \in \mathbb{Z} : n^3 + 2n = 3m + r, r \in \mathbb{Z}_{>0}^{<3}$$

- Chain of logic
  - Base case: $P(0)$ holds.
  - Inductive step: Assume $P(k)$ is true, prove $P(k+1)$ is true. Finally we have

    $$\forall k \geq 0, P(k) \Rightarrow P(k+1)$$

  - Conclusion: by the principle of induction, we have $\forall k \geq 0, P(k)$ is true.

§1.1.3. Strong Induction

Definition 1.1.3.1 (Strong Induction) **.**
Strong induction is a method of proof in which we prove that a statement is true for all natural numbers by proving that it is true for the smallest natural number and then proving that if it is true for all natural numbers less than or equal to some natural number, then it is true for the next natural number. Formally, let $P(n)$ be a statement for each $n \in \mathbb{N}_0$). If $P(0)$ is true, and $\forall k \in \mathbb{N}_0, (\forall m \leq k, P(m)) \Rightarrow P(k+1)$, then $\forall n \in \mathbb{N}, P(n)$ is true.

Example 1.1.3.1 .
Prove that every integer greater than 1 can be written as a product of prime numbers.

- Chain of logic:
  - Base case: $P(0)$ holds.
  - Inductive step: Assume $P(0), P(1), ..., P(k)$ are true, prove $P(k+1)$ is true. Finally we have

$$\forall k \geq 0, (\forall m \leq k, P(m)) \Rightarrow P(k+1) \text{ or}$$
$$\forall k \geq 0, (P(0) \wedge P(1) \wedge ... \wedge P(k)) \Rightarrow P(k+1) \text{ or}$$
$$\forall k \geq 0, \bigwedge_{m \leq k} P(m) \Rightarrow P(k+1)$$

§1.1.4. Generalised Induction

Definition 1.1.4.1 (Generalised Induction).
Generalised induction is a method of proof in which we prove that a statement is true for all objects in a set by proving that it is true for the smallest object and then proving that if it is true for all objects smaller than or equal to some object, then it is true for the next object. Formally, we implicitly define a bijection between the set of objects and the natural numbers, and then prove the statement for all natural numbers, as what we do in normal induction.

- pattern:
  - The conclusion holds when the structure is minimized.
  - Assume the conclusion holds for all smaller sub-structures, prove the conclusion holds for the current structure.
  - The conclusion holds for all structures.
- Why Generalize Induction?
  - Natural numbers are insufficient for complex structures (e.g., proving properties of graphs or programs).
  - Unified framework: Well-foundedness captures the essence of induction, allowing it to apply broadly.

§1.1.4.1. Structural Induction

Definition 1.1.4.1.1 (Structural Induction).
Structural induction is a generalisation of mathematical induction to data structures. It is a method of proof in which we prove that a property holds for all elements of a data structure by proving that it holds for the smallest elements and then proving that if it holds for all sub-structures of some element, then it holds for the element itself.

Remark 1.1.4.1.1 .
Using structual induction is always dependent on the recursive definition of the data structure we are working on.

In the context of computer science, we often use structural induction, which is a kind of generalised induction, to prove properties of data structures including strings, trees, and graphs.

Example 1.1.4.1.1 (Well-formed Parentheses).
A Well-formed string of parentheses is a string that consists of a series of opening and closing parentheses, such that each closing parenthesis matches the most recent unmatched opening parenthesis. For example, the strings "(()())" and "((()))" are well-formed, while the strings "())(" and "(()" are not well-formed. Prove that every well-formed string of parentheses has an equal number of opening and closing parentheses.

Remark 1.1.4.1.2.
Let $\Sigma = \{(,)\}$. Let Σ^* be the set of all finite strings over Σ. Let $\text{open} : \Sigma^* \to \mathbb{N}_0$ and $\text{close} : \Sigma^* \to \mathbb{N}_0$ be functions counting the number of opening and closing parentheses in a string S, respectively. ($\mathbb{N}_0$ denotes the set of non-negative integers). The set of **Well-Formed Parentheses** strings, denoted WFP, is a subset of Σ^* defined recursively as the smallest set satisfying:

[**Base Case:**] The empty string λ is in WFP.

[**Recursive Step 1:**] If S' is in WFP, then the string (S') is in WFP, where (S') is the concatenation of S' with an opening and closing parenthesis.

[**Recursive Step 2:**] If S' and T' are in WFP, then their concatenation $S'T'$ is in WFP.

Example 1.1.4.1.2.
Let Σ be a finite alphabet. A string over Σ is a finite sequence of characters from Σ recursively defined by the alphabet. We use Σ^* to denote the set of all strings over Σ. For some string $w \in \Sigma^*$, let w^R denote the reversed string of w. Prove that for any two strings $w_1, w_2 \in \Sigma^*$,

$$w_1^R w_2^R = (w_2 w_1)^R$$

Note: you may use λ to denote the empty string. Additionally, w^R is recursively defined as follows:
- If $w = \lambda$, then $w^R = \lambda$.
- If $w = xw'$, where $x \in \Sigma$ and $w' \in \Sigma^*$, then $w^R = w'^R x$.

- Other generalised induction(not covered in this lecture):
  - Transfinite Induction
  - Noetherian Induction

§2. Introduction to Graph Theory

Definition 2.1 (Graph).
A graph G is an ordered pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. Each edge is a pair of vertices (u, v), where $u, v \in V$. We use $V(G)$ and $E(G)$ to denote the set of vertices and edges of G, respectively.

- In a nutshell, a graph is an abstraction of some tangible or intangible objects and their relationships.
- The vertices represent the objects, and the edges represent the relationships between the objects.

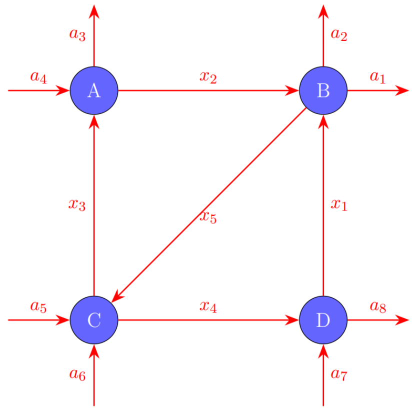


Figure 1: An example of a graph for road traffic

- The above is a Directed Graph, where the edges have directions.
- The graph is also weighted, meaning that the edges have weights, or attached with a value.
- Using the notation $G = (V, E)$, we can represent the graph as $G = (\{A, B, C, D\}, \{(A, B), (B, C), (C, A), (C, D), (D, B)\})$.

§2.1. Some Basic Notions

Definition 2.1.1 (Weight of Edges).
The weight of an edge in a graph is a value associated with the edge. The weight can represent the cost, distance, probability or any other value associated with the edge.

Definition 2.1.2 (Direction of Edges).
The edges of a graph can be directed or undirected. In a **directed graph**, the edges have directions, meaning that the edge (u, v) is different from the edge (v, u). In an **undirected graph**, the edges do not have directions, meaning that the edge (u, v) is the same as the edge (v, u).

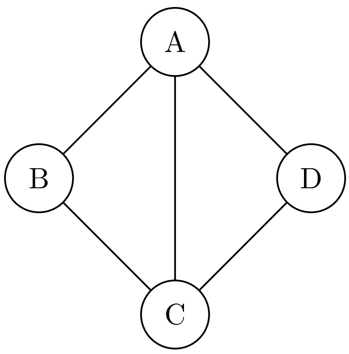


Figure 3: Undirected Graph

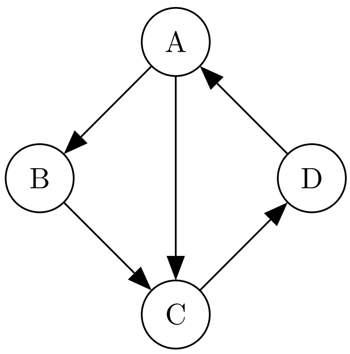


Figure 4: Directed Graph

Definition 2.1.3 (Loop).
A loop is an edge that connects a vertex to itself. In a graph, a loop is an edge of the form (v, v), where $v \in V$.

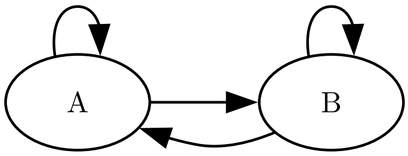


Figure 5: Graph with a loop

Definition 2.1.4 (Simple Graph).
A simple graph is a graph in which there is at most one edge between any two vertices and no edge from a vertex to itself.

Type	Edges	Multiple Edges?	Loops Allowed?
Simple graph	Undirected	No	No
Multigraph	Undirected	Yes	No
Pseudograph	Undirected	Yes	Yes
Simple directed graph	Directed	No	No
Directed multigraph	Directed	Yes	Yes
Mixed graph	Directed and undirected	Yes	Yes

Table 1: Graph Typology

§2.2. Applications of Graph

Graphs are widely used in computer science, mathematics, and other fields to model relationships between objects. Some common applications of graphs include:

- Social networks: representing relationships between people.
- Road networks: representing roads and intersections.
- Network traffic: representing data flow between devices.
- Computer networks: representing connections between computers.
- Scheduling: representing tasks and dependencies between tasks.
- Circuit design: representing components and connections between components.

Example 2.2.1.
An example of a graph for network traffic is the webpage ranking. Webpage ranking is an algorithm used by search engines to rank web pages in search results. The algorithm uses a graph to represent the web pages and the links between them. We cannot discuss the details of the algorithm here, but in this algorithm, we use a directed weighted graph to represent possibility of a user visiting a webpage from another webpage.

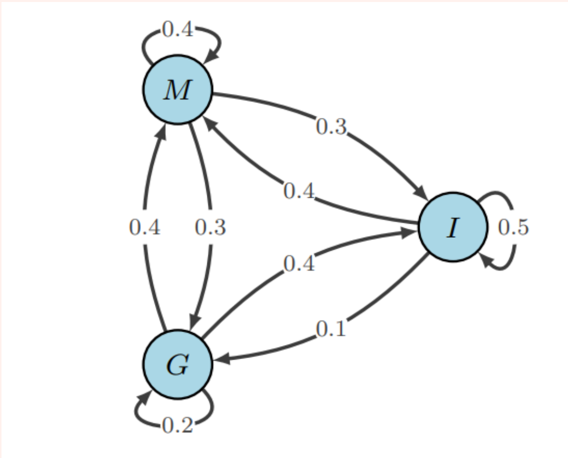


Figure 6: An example of a graph for network traffic

We will look into the details of this example in matrix representation of a graph.

§2.3. More Terminologies

Definition 2.3.1 (Degree of a Vertex).
The degree of a vertex v in a graph is the number of edges incident to v. In an undirected graph, the degree of a vertex is the number of edges connected to the vertex. In a directed graph, the degree of a vertex is the sum of the in-degree and out-degree of the vertex, where the in-degree is the number of edges pointing to the vertex, and the out-degree is the number of edges pointing from the vertex.

§2.3.1. Path and Cycle

Definition 2.3.1.1 (Path).
A path in a graph is a sequence of vertices in which each vertex is connected to the next vertex by an edge. A path is simple if it does not contain any repeated vertices.

For example, in the graph below, the sequence $A \to B \to C \to D$ is a path.

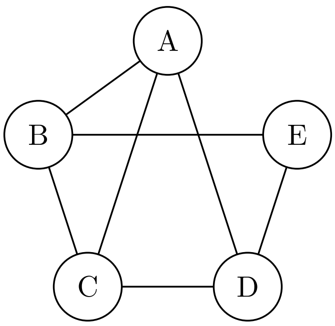


Figure 7: A graph with cycle

We also introduce some special subsets of paths.

Definition 2.3.1.2 (Cycle).
A cycle in a graph is a path that starts and ends at the same vertex. A cycle is simple if it does not contain any repeated vertices except the starting and ending vertex.

For example, in the previous graph, the sequence $A \to B \to C \to D \to A$ is a cycle.

§2.3.1.1. Euler Path and Circuit

Definition 2.3.1.1.1 (Euler Path).
An Euler path in a graph is a path that visits every edge exactly once. An Euler path may start and end at different vertices.

Definition 2.3.1.1.2 (Euler Circuit).
An Euler circuit in a graph is a cycle that visits every edge exactly once. An Euler circuit starts and ends at the same vertex.

Problem 2.3.1.1.1.
Can you find an Euler path/cycle in Figure 7?

Theorem 2.3.1.1.1 (Euler's Theorem).
Let $G = (V, E)$ be a finite, connected, undirected graph, then G:
- Has an Euler path if and only if it has exactly two or zero vertices of odd degree.
- Has an Euler circuit if and only if all vertices have even degree.

Lemma 2.3.1.1.1 (Maximum possible edges in a graph).
Let $G = (V, E)$ be a graph with n vertices.

If G is undirected, then the maximum number of edges in G is $\frac{|V|(|V|-1)}{2} = \frac{n(n-1)}{2}$.

If G is directed, then the maximum number of edges in G is $|V|(|V|-1) = n(n-1)$.

Proof. We use combinatorial proof techniques to prove this lemma.
- For an undirected graph, the number of edges is the number of ways to choose two vertices from V. This is given by the binomial coefficient $\binom{n}{2} = \frac{n(n-1)}{2}$.
- For a directed graph, the number of edges is the number of ways to choose an ordered pair of vertices from V. This is given by the product $n(n-1)$, by the rule of product from combinatorics.

□

§2.3.2. Connected & Ascyclic Graph

Definition 2.3.2.1 (Connected Graph).
A graph is connected if there is a path between every pair of vertices in the graph.

Definition 2.3.2.2 (Acyclic Graph).
A graph is acyclic if it does not contain any cycles.

Definition 2.3.2.3 (Complete Asyclic Graph).
A complete acyclic graph is a graph in which every pair of vertices is connected by a **unique** path.

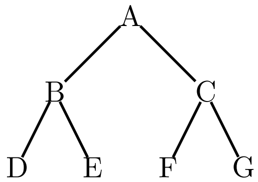


Figure 8: A tree with 7 vertices

Remark 2.3.2.1.
A more well-known name for a complete acyclic graph is a **tree**. Figure 8 is called a binary tree, as each vertex has at most two child nodes. It is also a complete or full binary tree, as all levels are fully filled except possibly for the last level, which is filled from left to right.

Problem 2.3.2.1.
Can you give a recursive definition of a tree? Thus, give a formal definition of a binary tree.

§2.3.2.1. Binary Tree

Binary tree is one of the most common tree structures in computer science.

We have already discussed the recursive definition of a binary tree. Here is another definition of a binary tree:

Definition 2.3.2.1.1 (Binary Tree).
A binary tree is a tree in which each node has at most two children, which are referred to as the left child and the right child.

The top vertex of the tree is called the **root** of the tree. The vertices that have no children are called **leaves** (node).

- A binary tree is called a **full binary tree** if each node has either zero or two children.
- A binary tree is called a **complete binary tree** if all levels are fully filled except possibly for the last level, which is filled from left to right.
- A binary tree is called a **perfect binary tree** if all internal nodes have two children and all leaves are at the same level.

Example 2.3.2.1.1.
The tree Figure 8 is a binary tree. It is also a complete binary tree, as all levels are fully filled except possibly for the last level, which is filled from left to right. Node A

Problem 2.3.2.1.1.
Prove that a tree with n vertices has $n-1$ edges.

Problem 2.3.2.1.2.
The level of a node in a tree is the number of edges on the path from the root to the leaf node. The root is at level 0.

For a binary tree of n nodes, the maximum height of the tree is $n-1$. Prove this proposition.

Problem 2.3.2.1.3.
Prove that at the i-th level of a binary tree, the maximum number of nodes is 2^i (counting from 0).

Problem 2.3.2.1.4.
If a binary tree of height h has l leaves, then $h \geq \lceil \log_2 l \rceil$, if the binary tree is full and balanced, then $h = \lceil \log_2 l \rceil$.

Prove this proposition.

§2.3.3. Subgraph

Definition 2.3.3.1 (Subgraph).
A subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. That is, a subgraph is a graph that contains a subset of the vertices and edges of the original graph.

In a straightforward way, a subgraph is a graph that can be obtained by removing some vertices and edges from the original graph.

§2.3.4. Complete Graph

Definition 2.3.4.1 (Complete Graph).
A complete graph is a graph in which every pair of vertices is connected by an edge. The number of edges in a complete graph with n vertices is

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

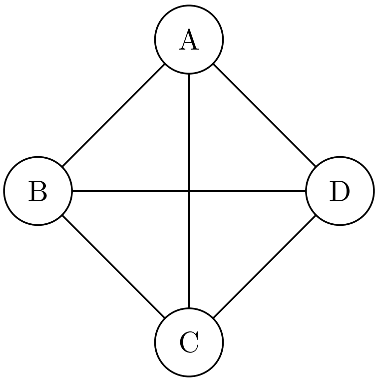


Figure 9: A complete graph with 4 vertices

Remark 2.3.4.1 .
A complete graph with n vertices is denoted by K_n.

§2.3.5. Bipartite Graph

Definition 2.3.5.1 (Bipartite Graph).
A bipartite graph is a graph whose vertices can be divided into two disjoint sets V_1 and V_2 such that every edge connects a vertex in V_1 to a vertex in V_2.

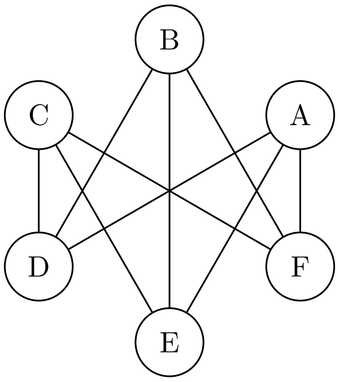


Figure 10: A bipartite graph with 3 vertices in each set

Notation 2.3.5.1 .
A complete bipartite graph, by convention, is denoted by $K_{\{m,n\}}$, where m and n are the number of vertices in the two sets V_1 and V_2, respectively.

For example, the graph above is denoted by $K_{\{3,3\}}$.

§2.3.6. Finite and Infinite Graph

Definition 2.3.6.1 (Finite Graph).
A graph is finite if it has a finite number of vertices and edges.

Definition 2.3.6.2 (Infinite Graph).
A graph is infinite if it has an infinite number of vertices or edges.

Remark 2.3.6.1.
The concept of infinite graphs is essential in mathematics and computer science, especially in the study of infinite structures and algorithms on infinite structures.

Problem 2.3.6.1.
Anything discussed earlier in the lecture can be abstract by a infinite graph?

§2.4. Useful Results on Graph

§2.4.1. Handshaking Theorem

Theorem 2.4.1.1 (Handshaking Theorem).
The handshaking theorem states that for any graph, the sum of the degrees of all vertices is equal to twice the number of edges. A loop at a vertex v is typically counted as contributing 2 to $\deg(v)$.

Formally, let $G = (V, E)$ be an undirected graph with $m = |E|$ edges. Then

$$\sum_{v \in V} \deg(v) = 2m.$$

How can we prove it? Eventhough it is trivial, we can prove it by structural induction on the definition of a undirected graph.

Problem 2.4.1.1.
There are 605 people in a party. Each person shakes hands with some other people. Suppose that each of them shakes hands with at least one person. Prove that there must be someone who shakes hands with at least two persons.

Theorem 2.4.1.2 (Number of Vertices with Odd Degree).
In any simple graph G, the number of vertices with odd degree is always even.

Problem 2.4.1.2.
Prove that among a group of people($n > 2$), there are at least 2 persons where the number of people they know is the same.

§2.5. Representation of Graph

Graph can be represented in different ways, including adjacency matrix, adjacency list, and incidence matrix. Each representation has its own advantages and disadvantages, and the choice of representation depends on the specific problem being solved. We introduce two of the most common representations: adjacency matrix and adjacency list.

§2.5.1. Adjacency Matrix

Definition 2.5.1.1 (Adjacency Matrix).
An adjacency matrix is a square matrix used to represent a graph. The rows and columns of the matrix correspond to the vertices of the graph, and the entries of the matrix indicate whether there is an edge between the corresponding vertices. If there is an edge between vertices u and v, the entry a_{uv} is 1; otherwise, it is 0.

If the graph is weighted, the entries of the adjacency matrix can be the weights of the edges instead of 0 or 1.

Formally, for a un weighted graph, the adjacency matrix A of a graph $G = (V, E)$ with n vertices is an $n \times n$ matrix defined as

$$A = (a_{ij}) = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix},$$

where $a_{ij} = 1$ if $(v_i, v_j) \in E$ and $a_{ij} = 0$ otherwise.

For a weighted graph, the adjacency matrix A of a graph $G = (V, E)$ with n vertices is an $n \times n$ matrix defined as

$$A = (a_{ij}) = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{pmatrix},$$

where w_{ij} is the weight of the edge (v_i, v_j) if $(v_i, v_j) \in E$ and $w_{ij} = 0$ otherwise.

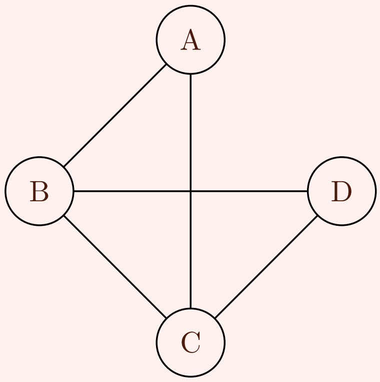
Example 2.5.1.1 .
Consider the graph $G = (V, E)$ with

$$V = \{A, B, C, D\}$$

and

$$E = \{(A, B), (B, C), (C, A), (C, D), (D, B)\}.$$

The adjacency matrix of G is



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Figure 11: A graph and its adjacency matrix

§2.5.1.1. Adjacency Matrix Power

Definition 2.5.1.1.1 (Matrix Power).
The power of a matrix A is defined as the matrix obtained by multiplying A by itself n times. The power of a matrix A to the nth power is denoted as A^n. For example, for a n dimensional square matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix},$$

The matrix A^2 is obtained by multiplying A by itself:

$$A^2 = A \cdot A = \begin{pmatrix} \sum_{k=1}^n a_{1k}a_{k1} & \sum_{k=1}^n a_{1k}a_{k2} & \dots & \sum_{k=1}^n a_{1k}a_{kn} \\ \sum_{k=1}^n a_{2k}a_{k1} & \sum_{k=1}^n a_{2k}a_{k2} & \dots & \sum_{k=1}^n a_{2k}a_{kn} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^n a_{nk}a_{k1} & \sum_{k=1}^n a_{nk}a_{k2} & \dots & \sum_{k=1}^n a_{nk}a_{kn} \end{pmatrix}$$

Example 2.5.1.1.1 (Matrix Power).
A simple example of 3 by 3 matrix A is

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The matrix A^2 is

$$A^2 = A \cdot A = \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}.$$

Let's decompose the calculation of each entry:

- The entry $a_{11} = a_{11}a_{11} + a_{12}a_{21} + a_{13}a_{31} = 1 \times 1 + 2 \times 4 + 3 \times 7 = 30$.

- The entry $a_{12} = a_{11}a_{12} + a_{12}a_{22} + a_{13}a_{32} = 1 \times 2 + 2 \times 5 + 3 \times 8 = 36$.
- The entry $a_{13} = a_{11}a_{13} + a_{12}a_{23} + a_{13}a_{33} = 1 \times 3 + 2 \times 6 + 3 \times 9 = 42$.
- The entry $a_{21} = a_{21}a_{11} + a_{22}a_{21} + a_{23}a_{31} = 4 \times 1 + 5 \times 4 + 6 \times 7 = 66$.
- The entry $a_{22} = a_{21}a_{12} + a_{22}a_{22} + a_{23}a_{32} = 4 \times 2 + 5 \times 5 + 6 \times 8 = 81$.
- The entry $a_{23} = a_{21}a_{13} + a_{22}a_{23} + a_{23}a_{33} = 4 \times 3 + 5 \times 6 + 6 \times 9 = 96$.
- The entry $a_{31} = a_{31}a_{11} + a_{32}a_{21} + a_{33}a_{31} = 7 \times 1 + 8 \times 4 + 9 \times 7 = 102$.
- The entry $a_{32} = a_{31}a_{12} + a_{32}a_{22} + a_{33}a_{32} = 7 \times 2 + 8 \times 5 + 9 \times 8 = 126$.
- The entry $a_{33} = a_{31}a_{13} + a_{32}a_{23} + a_{33}a_{33} = 7 \times 3 + 8 \times 6 + 9 \times 9 = 150$.

Problem 2.5.1.1.1 .
Consider the previous matrix, what is the trend of the matrix power A^n as n increases? That is to say, how do we evaluate

$$\lim_{n \to \infty} A^n.$$

Also think about:
- what is the adjacency matrix of graph in Example 2.2.1?
- What is the adjacency matrix of the graph in Figure 7? Work on it and try to find is there any interesting property of the matrix and its power.

Proposition 2.5.1.1.1 (Significance of Matrix Power).
For a simple undirected graph G, the matrix power M^n will have some interesting properties:
- the (i, j) entry of M^n is the number of paths of length n from vertex i to vertex j.
- the (i, i) entry of M^n is the number of circuits of length n starting and ending at vertex i.

Proof. We will prove the proposition by induction on n.

Base case: For $n = 1$, the matrix M^1 is the adjacency matrix of the graph G, and the (i, j) entry of M^1 is 1 if there is an edge between vertex i and vertex j and 0 otherwise. Therefore, the base case holds.

Inductive step: Assume the proposition holds for some $n = k$. We need to prove that the proposition holds for $n = k + 1$.

Let M^k be the matrix obtained by raising the adjacency matrix M to the kth power. The (i, j) entry of M^k is the number of paths of length k from vertex i to vertex j. The (i, i) entry of M^k is the number of circuits of length k starting and ending at vertex i.

The (i, j) entry of M^{k+1} is the sum of the products of the (i, k) entry of M^k and the (k, j) entry of M. Therefore, the (i, j) entry of M^{k+1} is the number of paths of length $k + 1$ from vertex i to vertex j. The (i, i) entry of M^{k+1} is the sum of the products of the (i, k) entry of M^k and the (k, i) entry of M. Therefore, the (i, i) entry of M^{k+1} is the number of circuits of length $k + 1$ starting and ending at vertex i. Therefore, the proposition holds for $n = k + 1$.

By the principle of mathematical induction, the proposition holds for all $n \in \mathbb{N}$. $\square$

§2.5.1.2. Implementation

```python
from typing import TypeVar, Generic, List, Optional, Tuple, Dict, Set, Any

V = TypeVar('V')  # Type for vertex identifiers
E = TypeVar('E')  # Type for edge weights (can be float, int, etc.)

class MatrixGraph(Generic[V, E]):
    """
    A graph implementation using an adjacency matrix.

    Handles directed/undirected and weighted/unweighted graphs.
    For unweighted graphs, the edge value '1' is used in the matrix.
    For weighted graphs, the edge weight is stored.
    'None' indicates the absence of an edge.

    Note: Adding/removing vertices is O(V^2) due to matrix resizing.
          Edge lookups/updates are O(1) after vertex index lookup.
          Space complexity is O(V^2).
          This representation is generally better for dense graphs.
    """

    def __init__(self, directed: bool = False, weighted: bool = False):
        """
        Initializes an empty graph.

        Args:
            directed: True if the graph is directed, False otherwise.
            weighted: True if the graph edges have weights, False otherwise.

        Complexity: O(1)
        """
        self._is_directed: bool = directed
        self._is_weighted: bool = weighted
        self._adj_matrix: List[List[Optional[E]]] = []
        self._vertex_to_index: Dict[V, int] = {}
        self._index_to_vertex: List[V] = []
        self._num_vertices: int = 0
        self._num_edges: int = 0

    def num_vertices(self) -> int:
        """
        Returns the number of vertices in the graph.
        Complexity: O(1)
        """
        return self._num_vertices
```

```python

    def num_edges(self) -> int:
        """
        Returns the number of edges in the graph.
        Complexity: O(1)
        """
        return self._num_edges

    def is_directed(self) -> bool:
        """
        Returns True if the graph is directed, False otherwise.
        Complexity: O(1)
        """
        return self._is_directed

    def is_weighted(self) -> bool:
        """
        Returns True if the graph is weighted, False otherwise.
        Complexity: O(1)
        """
        return self._is_weighted

    def get_vertices(self) -> List[V]:
        """
        Returns a list of all vertices in the graph.
        Complexity: O(V) - due to copying the list of vertices.
        """
        # Return a copy to prevent external modification of internal list
        return list(self._index_to_vertex)

    def has_vertex(self, vertex: V) -> bool:
        """
        Checks if a vertex exists in the graph.
        Complexity: O(1) average - dictionary lookup. O(V) worst case (highly unlikely with standard hash functions).
        """
        return vertex in self._vertex_to_index

    def _get_vertex_index(self, vertex: V) -> int:
        """
        Helper to get the index of a vertex, raising ValueError if not found.
        Complexity: O(1) average - dictionary lookup. O(V) worst case.
        """
        index = self._vertex_to_index.get(vertex)
        if index is None:
            raise ValueError(f"Vertex '{vertex}' not found in the graph.")
```

```
        return index

    def add_vertex(self, vertex: V) -> None:
        """
        Adds a vertex to the graph.

        If the vertex already exists, this method does nothing.
        Complexity: O(V^2) - dominated by matrix resizing (adding a row and column).
                      O(1) average if vertex already exists.

        Args:
            vertex: The vertex identifier to add.
        """
        if self.has_vertex(vertex): # O(1) avg
            return

        new_index = self._num_vertices
        self._vertex_to_index[vertex] = new_index # O(1) avg
        self._index_to_vertex.append(vertex)      # O(1) avg (amortized)

        # Resize matrix: Add a new column to existing rows - O(V)
        for row in self._adj_matrix:
            row.append(None) # O(1) for each row

        # Add a new row for the new vertex - O(V) creation + O(V) append = O(V)
        new_row = [None] * (self._num_vertices + 1)
        self._adj_matrix.append(new_row) # O(1) avg (amortized for list append)

        # Overall: O(1) + O(V) + O(V) = O(V), BUT standard analysis often assumes
        # matrix allocation/copying dominates, leading to O(V^2) practical view if
        # underlying memory needs reallocation or if we consider creating the new row/col elements.
        # Let's stick to O(V^2) as the most common analysis for matrix resizing.
        # If implemented without full reallocation (e.g., pre-allocating larger blocks),
        # it could be closer to O(V), but the standard matrix model implies V^2 for resizing.

        self._num_vertices += 1 # O(1)

    def remove_vertex(self, vertex: V) -> None:
        """
        Removes a vertex and all incident edges from the graph.
```

```

        Complexity: O(V^2) - dominated by creating the new smaller matrix and
                      recalculating edges. Index remapping is O(V).

        Args:
            vertex: The vertex identifier to remove.

        Raises:
            ValueError: If the vertex does not exist.
        """
        if not self.has_vertex(vertex): # O(1) avg
            raise ValueError(f"Vertex '{vertex}' not found for removal.")

        idx_to_remove = self._vertex_to_index[vertex] # O(1) avg

        # 1. Adjust edge count ( preliminary check, full recalc below) - O(V)
        # edges_removed = 0 ... (omitted for brevity, recalc is dominant)

        # 2. Remove vertex from mappings - O(1) avg dict del, O(V) list pop
        del self._vertex_to_index[vertex]
        self._index_to_vertex.pop(idx_to_remove)

        # 3. Remap indices - O(V)
        for v, i in self._vertex_to_index.items():
            if i > idx_to_remove:
                self._vertex_to_index[v] = i - 1

        # 4. Create a new smaller matrix - O(V^2)
        new_size = self._num_vertices - 1
        new_matrix = [[None for _ in range(new_size)] for _ in range(new_size)]
        current_new_row = 0
        for old_row_idx in range(self._num_vertices):
            if old_row_idx == idx_to_remove: continue
            current_new_col = 0
            for old_col_idx in range(self._num_vertices):
                if old_col_idx == idx_to_remove: continue
                new_matrix[current_new_row][current_new_col] = self._adj_matrix[old_row_idx][old_col_idx]
                current_new_col += 1
            current_new_row += 1

        self._adj_matrix = new_matrix # O(1) reference assignment
        self._num_vertices -= 1       # O(1)

        # 5. Recalculate edge count - O(V^2)
        self._recalculate_num_edges()
```

```python

    def _recalculate_num_edges(self):
        """
        Helper to recalculate the edge count by iterating the matrix.
        Complexity: O(V^2) - iterates through the entire matrix.
        """
        count = 0
        for r in range(self._num_vertices):
            for c in range(self._num_vertices):
                if self._adj_matrix[r][c] is not None:
                    if self._is_directed:
                        count += 1
                    else:
                        if r <= c: # Count undirected edges once
                            count += 1
        self._num_edges = count


    def add_edge(self, source: V, destination: V, weight: Optional[E] = 1) -> None:
        """
        Adds an edge between source and destination vertices.

        If the graph is unweighted, the weight parameter is ignored, and 1 is stored.
        If the graph is weighted, a weight must be provided (defaults to 1 if None).
        If the edge already exists, its weight is updated (if weighted).

        Complexity: O(1) average - after vertex index lookups (which are O(1) avg).

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.
            weight: The weight of the edge (used only if graph is weighted). Defaults to 1.

        Raises:
            ValueError: If source or destination vertices do not exist.
            ValueError: If the graph is weighted and weight is None.
        """
        u_idx = self._get_vertex_index(source) # O(1) avg
        v_idx = self._get_vertex_index(destination) # O(1) avg

        edge_value: Optional[E]
```

```
        if self._is_weighted:
            if weight is None:
                edge_value = 1 # Defaulting to 1
            else:
                edge_value = weight
        else:
            edge_value = 1 # Use 1 for unweighted graphs

        # Check if edge is newly added - O(1)
        is_new_edge = self._adj_matrix[u_idx][v_idx] is None

        # Matrix update - O(1)
        self._adj_matrix[u_idx][v_idx] = edge_value
        if not self._is_directed:
            # Additional matrix update - O(1)
            is_new_reverse = self._adj_matrix[v_idx][u_idx] is None if u_idx != v_idx else False
            self._adj_matrix[v_idx][u_idx] = edge_value
            # Edge count update - O(1)
            if is_new_edge or is_new_reverse:
                self._num_edges += 1
        elif is_new_edge: # Directed
            # Edge count update - O(1)
            self._num_edges += 1

    def remove_edge(self, source: V, destination: V) -> None:
        """
        Removes the edge between source and destination.

        If the graph is undirected, removes the edge in both directions.
        Does nothing if the edge doesn't exist.

        Complexity: O(1) average - after vertex index lookups (which are O(1) avg).

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.

        Raises:
            ValueError: If source or destination vertices do not exist.
        """
        u_idx = self._get_vertex_index(source) # O(1) avg
        v_idx = self._get_vertex_index(destination) # O(1) avg

        # Check existence - O(1)
```

```
        edge_existed = self._adj_matrix[u_idx][v_idx] is not None

        if edge_existed:
            # Matrix update(s) - O(1)
            self._adj_matrix[u_idx][v_idx] = None
            if not self._is_directed:
                 if u_idx != v_idx:
                     self._adj_matrix[v_idx][u_idx] = None
                 # Edge count update - O(1)
                 self._num_edges -= 1
            else: # Directed
                # Edge count update - O(1)
                self._num_edges -= 1


    def has_edge(self, source: V, destination: V) -> bool:
        """
        Checks if an edge exists between source and destination.

        Complexity: O(1) average - after vertex index lookups (which are O(1) avg).

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.

        Returns:
            True if the edge exists, False otherwise.

        Raises:
            ValueError: If source or destination vertices do not exist.
        """
        u_idx = self._get_vertex_index(source) # O(1) avg
        v_idx = self._get_vertex_index(destination) # O(1) avg
        # Matrix access - O(1)
        return self._adj_matrix[u_idx][v_idx] is not None

    def get_edge_data(self, source: V, destination: V) -> Optional[E]:
        """
        Gets the weight or data associated with the edge.

        Returns None if the edge does not exist.
        For unweighted graphs, returns 1 if the edge exists.

        Complexity: O(1) average - after vertex index lookups (which are O(1) avg).
```

```

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.

        Returns:
            The edge weight/data, or None if the edge doesn't exist.

        Raises:
            ValueError: If source or destination vertices do not exist.
        """
        u_idx = self._get_vertex_index(source) # O(1) avg
        v_idx = self._get_vertex_index(destination) # O(1) avg
        # Matrix access - O(1)
        return self._adj_matrix[u_idx][v_idx]

    def get_neighbors(self, vertex: V) -> List[Tuple[V, Optional[E]]]:
        """
        Gets a list of neighbors for a given vertex, along with edge data.

        Complexity: O(V) - Must iterate through a full row of the matrix.

        Args:
            vertex: The vertex identifier.

        Returns:
            A list of tuples, where each tuple is (neighbor_vertex, edge_weight).
            Returns an empty list if the vertex has no neighbors.

        Raises:
            ValueError: If the vertex does not exist.
        """
        u_idx = self._get_vertex_index(vertex) # O(1) avg
        neighbors = []
        # Iterate through row - O(V)
        for v_idx in range(self._num_vertices):
            edge_data = self._adj_matrix[u_idx][v_idx] # O(1) access
            if edge_data is not None:
                neighbor_vertex = self._index_to_vertex[v_idx] # O(1) access
                neighbors.append((neighbor_vertex, edge_data)) # O(1) avg append
        return neighbors

    def get_edges(self) -> List[Tuple[V, V, Optional[E]]]:
        """
```

```
        Gets a list of all edges in the graph.

        For undirected graphs, each edge is listed only once (e.g., (u, v, w) but not (v, u, w)).

        Complexity: O(V^2) - Must iterate through the relevant portion of the matrix
                     (whole matrix for directed, roughly half for undirected).

        Returns:
            A list of tuples, where each tuple is (source_vertex, destination_vertex, edge_weight).
        """
        edges = []
        # Nested loops iterate O(V^2) times (or O(V^2/2) for undirected)
        for u_idx in range(self._num_vertices):
            start_j = u_idx if not self._is_directed else 0
            for v_idx in range(start_j, self._num_vertices):
                edge_data = self._adj_matrix[u_idx][v_idx] # O(1) access
                if edge_data is not None:
                    source = self._index_to_vertex[u_idx]      # O(1) access
                    destination = self._index_to_vertex[v_idx] # O(1) access
                    edges.append((source, destination, edge_data)) # O(1) avg append
        return edges

    def __len__(self) -> int:
        """
        Returns the number of vertices in the graph.
        Complexity: O(1)
        """
        return self.num_vertices()
```

§2.5.2. Adjacency List

Definition 2.5.2.1 (Adjacency List).
An adjacency list is a collection of lists used to represent a graph. Each list in the collection corresponds to a vertex of the graph, and the elements of the list are the vertices adjacent to the corresponding vertex. In an undirected graph, the adjacency list of a vertex v contains all the vertices adjacent to v. In a directed graph, the adjacency list of a vertex v contains all the vertices that have an edge pointing to v.

Example 2.5.2.1.
Consider the graph $G = (V, E)$ with

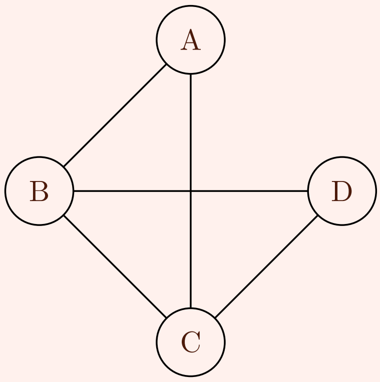
$$V = \{A, B, C, D\}$$

and

$$E = \{(A, B), (B, C), (C, A), (C, D), (D, B)\}.$$

The adjacency list of G is

$$A : \{B, C\}, B : \{A, C, D\}, C : \{A, B, D\}, D : \{B\}.$$



Node	Adjacency List
A	{B, C}
B	{A, C, D}
C	{A, B, D}
D	{B}

Table 2: A graph and its adjacency list

§2.5.2.1. Implementation

Remark 2.5.2.1.1 .
Do note that, the adjacency list attached to each vertex can be implemented as a list, set, or any other data structure that supports efficient insertion and deletion of elements. The choice of data structure depends on the specific problem being solved. In our occasion, we use dictionaries to represent the adjacency list:

```
[Vertex, Dict[Neighbor, EdgeWeight]]
```

Python

```python
from typing import TypeVar, Generic, List, Optional, Tuple, Dict, Set, Any, Iterable, Hashable

V = TypeVar('V', bound=Hashable)  # Type for vertex identifiers (must be hashable)
E = TypeVar('E')                  # Type for edge weights (can be float, int, etc.)

class ListGraph(Generic[V, E]):
    """
    A graph implementation using adjacency lists represented by dictionaries.
    The structure is: {vertex: {neighbor: weight/data, ...}, ...}

    Handles directed/undirected and weighted/unweighted graphs.
    For unweighted graphs, the edge value '1' (or True) is often used, here we use 1.
    'None' edge data is possible if explicitly added but generally avoided.

```

```
    Note: Vertex addition/lookup is O(1) avg. Edge addition/lookup is O(1) avg.
          Getting neighbors is O(degree(V)). Vertex removal is O(V + E) in the worst case
          (must check all other vertices for incoming edges). Space complexity is O(V + E).
          This representation is generally better for sparse graphs.
    """

    def __init__(self, directed: bool = False, weighted: bool = False):
        """
        Initializes an empty graph.

        Args:
            directed: True if the graph is directed, False otherwise.
            weighted: True if the graph edges have weights, False otherwise.

        Complexity: O(1)
        """
        self._is_directed: bool = directed
        self._is_weighted: bool = weighted

        # The core adjacency list structure: Dict[Vertex, Dict[Neighbor, EdgeData]]
        self._adj_list: Dict[V, Dict[V, Optional[E]]] = {}

        self._num_vertices: int = 0
        self._num_edges: int = 0

    def num_vertices(self) -> int:
        """
        Returns the number of vertices in the graph.
        Complexity: O(1)
        """
        # Alternatively, could use len(self._adj_list), but keeping a counter is safer
        return self._num_vertices

    def num_edges(self) -> int:
        """
        Returns the number of edges in the graph.
        Complexity: O(1)
        """
        return self._num_edges

    def is_directed(self) -> bool:
        """
```

```python
        Returns True if the graph is directed, False otherwise.
        Complexity: O(1)
        """
        return self._is_directed

    def is_weighted(self) -> bool:
        """
        Returns True if the graph is weighted, False otherwise.
        Complexity: O(1)
        """
        return self._is_weighted

    def get_vertices(self) -> List[V]:
        """
        Returns a list of all vertices in the graph.
        Complexity: O(V) - Collects keys from the dictionary.
        """
        return list(self._adj_list.keys())

    def has_vertex(self, vertex: V) -> bool:
        """
        Checks if a vertex exists in the graph.
        Complexity: O(1) average - dictionary key lookup.
        """
        return vertex in self._adj_list

    def add_vertex(self, vertex: V) -> None:
        """
        Adds a vertex to the graph.

        If the vertex already exists, this method does nothing.
        Complexity: O(1) average - dictionary insertion/check.

        Args:
            vertex: The vertex identifier to add. Must be hashable.
        """
        if vertex not in self._adj_list: # O(1) avg check
            self._adj_list[vertex] = {}  # Add vertex with an empty neighbor dict - O(1) avg insert
            self._num_vertices += 1 # O(1)

    def remove_vertex(self, vertex: V) -> None:
        """
        Removes a vertex and all incident edges from the graph.

```

```
        Complexity: O(V + E) worst case. O(V + degree(vertex)) average if graph is sparse.
                    Need to iterate through all vertices potentially (O(V)) to remove
                    incoming edges. Each incoming edge removal is O(1) avg.
                    Removing outgoing edges is O(degree(vertex)).

        Args:
            vertex: The vertex identifier to remove.

        Raises:
            ValueError: If the vertex does not exist.
        """
        if not self.has_vertex(vertex): # O(1) avg check
            raise ValueError(f"Vertex '{vertex}' not found for removal.")

        # 1. Count and remove outgoing edges from the vertex being removed
        # Complexity: O(degree(vertex)) to count, affects edge counter later
        outgoing_edges_data = self._adj_list[vertex]
        outgoing_edge_count = len(outgoing_edges_data) # O(1) for dict len usually, but conceptually O(degree) related work

        # 2. Remove the vertex itself and its outgoing edges list from the main dictionary
        # Complexity: O(1) average deletion
        del self._adj_list[vertex]
        self._num_vertices -= 1 # O(1)

        # 3. Remove all incoming edges pointing to the removed vertex
        # Complexity: O(V + E_in) worst case where E_in is incoming edges, O(V) average for sparse graphs
        # Need to check every *other* vertex's neighbor list
        incoming_removed_count = 0
        vertices_to_check = list(self._adj_list.keys()) # O(V) to create list
        for u in vertices_to_check: # O(V) iteration
            if vertex in self._adj_list[u]: # O(1) avg check
                del self._adj_list[u][vertex] # O(1) avg deletion
                incoming_removed_count += 1 # O(1)

        # 4. Adjust edge count carefully based on directedness
        # O(1) operations
        if self._is_directed:
            # Total edges removed = outgoing + incoming
            self._num_edges -= (outgoing_edge_count + incoming_removed_count)
        else: # Undirected
            # Each edge (u, v) was represented as u->v and v->u.
```

```
            # Removing vertex 'v' deleted its entry (v -> neighbors), accounting for outgoing_edge_count edges.
            # Iterating through others deleted incoming edges (u -> v).
            # Since each edge is counted once, the total number of edges removed is simply outgoing_edge_count.
            # (The incoming_removed_count should equal outgoing_edge_count if implemented correctly).
            self._num_edges -= outgoing_edge_count


    def add_edge(self, source: V, destination: V, weight: Optional[E] = 1) -> None:
        """
        Adds an edge between source and destination vertices.

        If the graph is unweighted, the weight parameter is ignored and 1 is stored.
        If the graph is weighted, the provided weight is stored (defaults to 1 if None).
        If the edge already exists, its weight is updated.

        Complexity: O(1) average - Dictionary lookups and insertions.

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.
            weight: The weight of the edge (used only if graph is weighted). Defaults to 1.

        Raises:
            ValueError: If source or destination vertices do not exist.
        """
        # Ensure vertices exist first - O(1) avg each
        if source not in self._adj_list:
            raise ValueError(f"Source vertex '{source}' not found.")
        if destination not in self._adj_list:
            raise ValueError(f"Destination vertex '{destination}' not found.")

        edge_value: Optional[E]
        if self._is_weighted:
            edge_value = weight if weight is not None else 1 # Default weight 1 if None - O(1)
        else:
            edge_value = 1 # Use 1 for unweighted graphs - O(1)

        # Check if edge is new before adding/updating - O(1) avg
        is_new_edge = destination not in self._adj_list[source]
```

```python

        # Add/update the edge - O(1) avg
        self._adj_list[source][destination] = edge_value

        # Handle edge count and undirected case
        if self._is_directed:
            if is_new_edge:
                self._num_edges += 1 # O(1)
        else: # Undirected
            # Also add/update the reverse edge, but only if not a self-loop
            if source != destination:
                 # Check if reverse is new *before* adding/updating it - O(1) avg
                is_new_reverse = source not in self._adj_list[destination]
                 # Add/update reverse edge - O(1) avg
                self._adj_list[destination][source] = edge_value
            else:
                is_new_reverse = False # Self-loop already handled by is_new_edge

            # Increment count only once if the edge was truly new (not just an update)
            # For undirected, this means it didn't exist in *either* direction before.
            # In our implementation, is_new_edge covers the source->dest check.
            # If it's a self-loop, is_new_reverse is False.
            # If not a self-loop, is_new_reverse checks dest->source.
            # An edge is conceptually new if it wasn't present in the canonical direction (e.g., source->dest).
            if is_new_edge:
                 self._num_edges += 1 # O(1)

    def remove_edge(self, source: V, destination: V) -> None:
        """
        Removes the edge between source and destination.

        If the graph is undirected, removes the edge in both directions.
        Does nothing if the edge doesn't exist.

        Complexity: O(1) average - Dictionary lookups and deletions.

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.

        Raises:
            ValueError: If source or destination vertices do not exist.
```

```
        """
        # Check vertex existence - O(1) avg each
        if source not in self._adj_list:
            raise ValueError(f"Source vertex '{source}' not found.")
        if destination not in self._adj_list:
            raise ValueError(f"Destination vertex '{destination}' not found.")

        # Check if edge exists before trying to delete - O(1) avg
        if destination in self._adj_list[source]:
            del self._adj_list[source][destination] # O(1) avg deletion
            edge_was_removed = True
        else:
            edge_was_removed = False

        # If edge existed and was removed, decrement count and handle undirected
        if edge_was_removed:
            self._num_edges -= 1 # O(1)

            # If undirected, remove the reverse edge as well (if not a self-loop)
            if not self._is_directed and source != destination:
                # Check existence before deleting reverse for robustness - O(1) avg
                if source in self._adj_list[destination]:
                    del self._adj_list[destination][source] # O(1) avg deletion
                # Do NOT decrement edge count again for undirected


    def has_edge(self, source: V, destination: V) -> bool:
        """
        Checks if an edge exists between source and destination.

        Complexity: O(1) average - Dictionary lookups.

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.

        Returns:
            True if the edge exists, False otherwise. Returns False if vertices don't exist.
        """
        if source not in self._adj_list: # O(1) avg check
            return False
        # Check if destination is a key in the source's neighbor dictionary - O(1) avg
```

```python
        return destination in self._adj_list[source]

    def get_edge_data(self, source: V, destination: V) -> Optional[E]:
        """
        Gets the weight or data associated with the edge.

        Returns None if the edge or vertices do not exist.
        For unweighted graphs, returns 1 if the edge exists.

        Complexity: O(1) average - Dictionary lookups.

        Args:
            source: The source vertex identifier.
            destination: The destination vertex identifier.

        Returns:
            The edge weight/data, or None if the edge/vertex doesn't exist.
        """
        if source not in self._adj_list: # O(1) avg check
            return None
        # dict.get() returns None if key (destination) not found - O(1) avg
        return self._adj_list[source].get(destination)

    def get_neighbors(self, vertex: V) -> Iterable[Tuple[V, Optional[E]]]:
        """
        Gets an iterable of neighbors for a given vertex, along with edge data.

        Complexity: O(degree(vertex)) - Iterates through the vertex's neighbor dictionary items.
                     Getting the items view itself is O(1).

        Args:
            vertex: The vertex identifier.

        Returns:
            An iterable view (dict_items) of tuples (neighbor_vertex, edge_weight).
            Use list() around the result if a list copy is needed.

        Raises:
            ValueError: If the vertex does not exist.
        """
        if vertex not in self._adj_list: # O(1) avg check
            raise ValueError(f"Vertex '{vertex}' not found.")

```

```
        # .items() provides an efficient view for iteration - O(1) to create view
        # Iteration over the view takes O(degree(vertex))
        return self._adj_list[vertex].items()

    def get_edges(self) -> List[Tuple[V, V, Optional[E]]]:
        """
        Gets a list of all edges in the graph.

        For undirected graphs, each edge is listed only once (conventionally).
        Uses a set to track visited pairs for undirected graphs to avoid duplicates.

        Complexity: O(V + E) - Iterates through all vertices (O(V)) and then through
                     all neighbors for each vertex (totaling O(E) across all vertices).
                     Set operations are O(1) average.

        Returns:
            A list of tuples, where each tuple is (source_vertex, destination_vertex, edge_weight).
        """
        edges = []
        # Used only for undirected graphs to avoid adding (u,v) and (v,u)
        visited_undirected_pairs = set() # O(1) space overhead initially

        # O(V) outer loop
        for source, neighbors in self._adj_list.items():
            # O(degree(source)) inner loop -> totals O(E) over the outer loop
            for destination, weight in neighbors.items():
                if self._is_directed:
                    edges.append((source, destination, weight)) # O(1) avg append
                else:
                    # Ensure undirected edges are added only once
                    # Create a canonical representation (frozenset for hashability regardless of order)
                    edge_pair = frozenset((source, destination)) # O(1) creation (assuming hashable V)
                    if edge_pair not in visited_undirected_pairs: # O(1) avg set lookup
                        edges.append((source, destination, weight)) # O(1) avg append
                        visited_undirected_pairs.add(edge_pair) # O(1) avg set add
        return edges # O(E) size list returned

```

```python
    def __len__(self) -> int:
        """
        Returns the number of vertices in the graph.
        Allows `len(graph)` syntax.
        Complexity: O(1)
        """
        return self.num_vertices()
```

§2.5.3. Trade-offs between Adjacency Matrix and Adjacency List

Definition 2.5.3.1 (Sparse and Dense Graph).
A graph is considered sparse if the number of edges is much less than the number of vertices squared, i.e., $|E| \ll |V|^2$. Sparse graphs have relatively few edges compared to the number of possible edges.

Accordingly, a graph is considered dense if the number of edges is close to the number of vertices squared, i.e., $|E| \sim |V|^2$. Dense graphs have many edges compared to the number of possible edges.

Feature / Operation	Adjacency Matrix (`MatrixGraph`)	Adjacency List (`ListGraph`)	Winner / Notes
Space Complexity	$O(V^2)$	$O(V+E)$	List (especially for sparse graphs)
Add Edge	$O(1)$ avg	$O(1)$ avg	Tie (both fast on average)
Remove Edge	$O(1)$ avg	$O(1)$ avg	Tie (both fast on average)
Check Edge / Get Weight	$O(1)$ avg	$O(1)$ avg	Tie (both fast on average)
Get Neighbors(u)	$O(V)$	$O(\deg(u))$	List (much faster for sparse graphs)
Get All Edges	$O(V^2)$	$O(V+E)$	List (much faster for sparse graphs)
Add Vertex	$O(V)$ or $O(V^2)$ if resizing	$O(1)$ avg	List (much faster & simpler)
Remove Vertex	$O(V^2)$	$O(V+E)$ or $O(V)$ avg	List (generally better)

Table 3: Comparison of Adjacency Matrix and Adjacency List Graph Implementations

§2.6. Basic Graph Algorithms

We discuss two fundamental graph traversal algorithms: Breadth-First Search (BFS) and Depth-First Search (DFS). These algorithms are used to explore and search a graph, and they can be applied to both directed and undirected graphs.

§2.6.1. Breadth-First Search

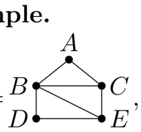
§2.6.1.1. Definition

Definition 2.6.1.1.1 (Breadth-First Search (BFS)).
Breadth-First Search (BFS) is a graph traversal algorithm that explores the graph level by level. Starting from a source vertex, BFS visits all its neighbors first, then the neighbors' neighbors, and so on. The algorithm uses a **queue** to keep track of the vertices to visit next.

Algorithm 1: Breadth-First Search

```
function BFS(graph, initial_vertex):
    visited = set()
    queue = [initial_vertex]
    visited.add(initial_vertex)
    while queue is not empty:
        current_vertex = queue.dequeue()
        for neighbor in graph.get_neighbors(current_vertex):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.enqueue(neighbor)
    return visited
```

Example.

If $G =$ 

, with root vertex A.

Then Q and T grow as follows:

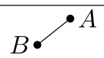
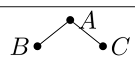
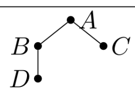
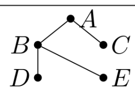
Step	Q	T
1	A	
2	AB	
3	ABC	
4	BC	
5	BCD	
6	$BCDE$	
7	CDE	
8	DE	
9	E	

Figure 13: Breadth-First Search (BFS) traversal of a graph starting from vertex A.

§2.6.1.2. Complexity Analysis

§2.6.1.2.1. Space Complexity

For a graph with V vertices and E edges:

- If implemented by adjacency matrix, the space complexity of Breadth-First Search (BFS) is $O(V)$. This is because:
  - we maintain a queue of vertices to visit, which can contain all vertices in the worst case, contributing $O(V)$ auxilary space.
  - we maintain a set of visited vertices, which can contain all vertices in the worst case, contributing $O(V)$ auxilary space.
- If implemented by adjacency list, the space complexity of Breadth-First Search (BFS) is also $O(V)$. The reason is similar to the adjacency matrix case, but the space usage is more efficient for sparse graphs.

§2.6.1.2.2. Time Complexity

Definition 2.6.1.2.2.1 (Big-O Notation).
By assuming the input of an algorithm is of variable size n, we use big-O notation to analyse the assymptotic upper bound of the time complexity when $n \to \infty$. We state that, in the worst case, the time complexity $T(n)$ of an algorithm with input size n has $T(n) = O(g(n))$, if

$$\exists (c > 0 \land r \in \mathbb{N}^{+}) \text{ such that } \forall n > r, 0 \leq f(n) \leq c \cdot g(n),$$

where $f(n)$ is the exact time complexity of the algorithm, and $g(n)$ is the upper bound of the time complexity.

For a graph with V vertices and E edges:

- If implemented by adjacency matrix, the time complexity of Breadth-First Search (BFS) is $O(V^2)$. This is because:
  - The queue operations (enqueue and dequeue) will be excecuted for each vertex, contributing $O(V)$ time.
  - The loop to visit neighbors of each vertex will be executed for each vertex. For adjacency matrix, we will run a outer loop of $|V|$ and inner loop of $|V|$ in the worst case, contributing $O(V^2)$ time.
- If implemented by adjacency list, the time complexity of Breadth-First Search (BFS) is $O(V + E)$. This is because:
  - The queue operations (enqueue and dequeue) are will be excecuted for each vertex, contributing $O(V)$ time.
  - The loop to visit neighbors of each vertex will be executed for each vertex. For adjacency list, we will run a outer loop of $|V|$ and inner loop of $\deg(V)$ in the worst case, and $\max(\deg(V)) = E$ in the worst case, contributing $O(V + E)$ time.

§2.6.2. Depth-First Search

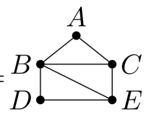
§2.6.2.1. Definition

Definition 2.6.2.1.1 (Depth-First Search (DFS)).
Depth-First Search (DFS) is a graph traversal algorithm that explores the graph by going as deep as possible along each branch before backtracking. Starting from a source vertex,

DFS visits the first neighbor, then the neighbor's neighbor, and so on until it reaches a dead-end. The algorithm uses a **stack** to keep track of the vertices to visit next.

Algorithm 2: Depth-First Search

```
function DFS(graph, initial_vertex):
    visited = set()
    stack = [initial_vertex]
    while stack is not empty:
        current_vertex = stack.pop()
        if current_vertex not in visited:
            visited.add(current_vertex)
            for neighbor in graph.get_neighbors(current_vertex):
                if neighbor not in visited:
                    stack.push(neighbor)
    return visited
```

$G =$ , with root vertex A.

Example.
We use the same G, and take the top of S to its right hand end.

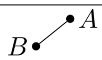
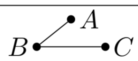
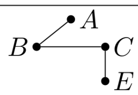
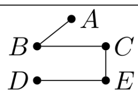
Step	S	T
1	A	
2	AB	
3	ABC	
4	$ABCE$	
4	$ABCED$	
6	$ABCE$	
7	ABC	
8	AB	
9	A	

Figure 14: Depth-First Search (DFS) traversal of a graph starting from vertex A.

§2.6.2.2. Complexity Analysis

§2.6.2.2.1. Space Complexity

For a graph with V vertices and E edges:

- If implemented by adjacency matrix, the space complexity of Depth-First Search (DFS) is $O(V)$. This is because:
  - we maintain a stack of vertices to visit, which can contain all vertices in the worst case, contributing $O(V)$ auxilary space.
  - we maintain a set of visited vertices, which can contain all vertices in the worst case, contributing $O(V)$ auxilary space.
- If implemented by adjacency list, the space complexity of Depth-First Search (DFS) is also $O(V)$. The reason is similar to the adjacency matrix case, but the space usage is more efficient for sparse graphs.

§2.6.2.2.2. **Time Complexity**

For a graph with V vertices and E edges:

- If implemented by adjacency matrix, the time complexity of Depth-First Search (DFS) is $O(V^2)$. This is because:
  - The stack operations (push and pop) are will be excecuted for each vertex, contributing $O(V)$ time in the worst case.
  - The loop to visit neighbors of each vertex will be executed for each vertex. For adjacency matrix, we will run a outer loop of $|V|$ and inner loop of $|V|$ in the worst case, contributing $O(V^2)$ time.
- If implemented by adjacency list, the time complexity of Depth-First Search (DFS) is $O(V + E)$. This is because:
  - The stack operations (push and pop) are will be excecuted for each vertex, contributing $O(V)$ time.
  - The loop to visit neighbors of each vertex will be executed for each vertex. For adjacency list, we iterate through all vertices by accessing V times of $O(1)$ operetion in the hash table, and iterate through all edges by accessing E times of $O(1)$ operation in the inner hash table, contributing $O(V + E)$ time.