

UNIT 10

Graphs

(5 Hours) — Data Structures and Algorithms

A graph is the most general non-linear data structure: unlike a tree, it places no restriction on how its elements may be connected. This unit introduces graph terminology and the graph ADT, the transitive closure and Warshall's algorithm, the main types of graph, traversal and spanning forests, Kruskal's algorithm for minimum spanning trees, and shortest-path computation using the greedy approach embodied in Dijkstra's algorithm.

10.1 Introduction to Graphs

A graph G is defined as an ordered pair $G = (V, E)$, where V is a finite non-empty set of vertices (also called nodes) and E is a set of edges, each edge being a pair of vertices. Unlike a tree, a graph has no root, may contain cycles, and a vertex may be reached by many different paths — or by none at all.

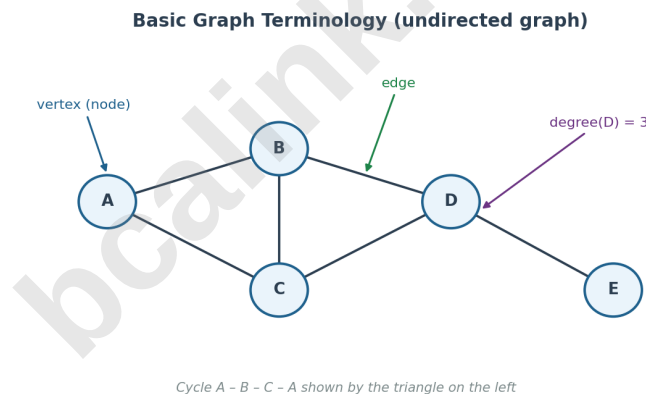


Fig 10.1 — An undirected graph showing vertices, edges, degree and a cycle.

10.1.1 Basic Terminology

Term	Meaning
Vertex (node)	A fundamental element of the graph, holding data
Edge	A connection between a pair of vertices
Adjacent vertices	Two vertices joined directly by an edge
Incident edge	An edge is incident on the two vertices it connects
Degree of a vertex	The number of edges incident on it
In-degree / Out-degree	In a digraph: edges coming into / going out of a vertex

Term	Meaning
Path	A sequence of vertices in which each consecutive pair is joined by an edge
Simple path	A path in which no vertex is repeated
Cycle	A path that starts and ends at the same vertex
Connected graph	A graph in which a path exists between every pair of vertices
Component	A maximal connected portion of a graph
Loop (self-edge)	An edge that joins a vertex to itself

Note: A tree is simply a special case of a graph: a connected, undirected graph with no cycles and exactly $n - 1$ edges for n vertices.

10.1.2 Applications of Graphs

- Road and transport networks — cities as vertices, roads as weighted edges.
- Computer networks — routers as vertices, communication links as edges.
- Social networks — people as vertices, friendships or follows as edges.
- Web page ranking — pages as vertices, hyperlinks as directed edges.
- Task scheduling — tasks as vertices and dependencies as directed edges in a DAG.
- Circuit design, map colouring, and course prerequisite planning.

10.2 Graph as an Abstract Data Type (ADT)

The graph ADT specifies the operations a graph must support, independently of how it is stored in memory.

Operation	Description
createGraph()	Creates an empty graph with no vertices or edges
insertVertex(v)	Adds a new vertex v to the graph
insertEdge(u, v)	Adds an edge connecting vertices u and v
deleteVertex(v)	Removes vertex v together with all edges incident on it
deleteEdge(u, v)	Removes the edge between u and v
isAdjacent(u, v)	Returns true if an edge joins u and v
neighbours(v)	Returns the list of all vertices adjacent to v
traverse(v)	Visits every vertex reachable from v exactly once

10.2.1 Representation of Graphs

There are two standard ways to store a graph in memory, and the choice has a large effect on both memory use and operation speed.

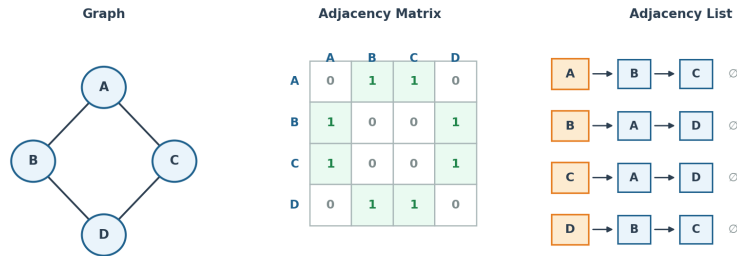


Fig 10.2 — The same graph stored as an adjacency matrix and as an adjacency list.

- **Adjacency Matrix** — a two-dimensional array A of size $n \times n$, where $A[i][j] = 1$ if an edge joins vertex i to vertex j , and 0 otherwise. For a weighted graph the weight is stored instead of 1.
- **Adjacency List** — an array of n linked lists, where the list for vertex i holds all vertices adjacent to i .

Aspect	Adjacency Matrix	Adjacency List
Space required	$O(n^2)$ always	$O(n + e)$ for e edges
Check if edge (u,v) exists	$O(1)$ — direct array access	$O(\text{degree of } u)$
List all neighbours of v	$O(n)$ — must scan a whole row	$O(\text{degree of } v)$
Add an edge	$O(1)$	$O(1)$
Best suited for	Dense graphs (e close to n^2)	Sparse graphs (e much less than n^2)

Note: For an undirected graph the adjacency matrix is always symmetric, since edge (u, v) is the same as edge (v, u) .

10.3 Types of Graph

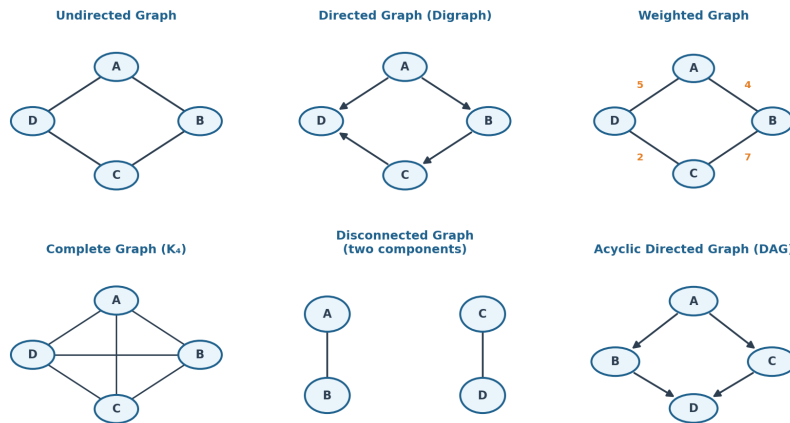


Fig 10.3 — The main types of graph.

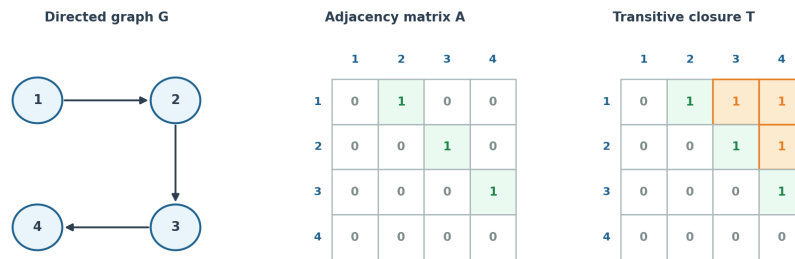
Type	Description
Undirected graph	Edges have no direction; edge (u, v) is the same as (v, u)
Directed graph (digraph)	Every edge has a direction, from a source vertex to a destination vertex
Weighted graph	Each edge carries a numeric weight representing cost, distance or capacity
Complete graph	Every pair of distinct vertices is joined by an edge; K_n has $n(n - 1)/2$ edges
Connected graph	A path exists between every pair of vertices
Disconnected graph	Splits into two or more separate components
Cyclic graph	Contains at least one cycle
Acyclic graph	Contains no cycle; a directed acyclic graph is called a DAG
Simple graph	Has no loops and no more than one edge between any pair of vertices
Multigraph	Permits multiple (parallel) edges between the same pair of vertices
Sparse / Dense graph	Few edges relative to n^2 / close to the maximum possible number of edges

10.4 Transitive Closure and Warshall's Algorithm

10.4.1 Transitive Closure

The adjacency matrix tells us whether one vertex reaches another in a single step. Often we need to know whether vertex i can reach vertex j by any path, of any length. This reachability information is called the transitive closure of the graph.

Formally, the transitive closure of G is a matrix T where $T[i][j] = 1$ if there exists a path of length one or more from vertex i to vertex j , and 0 otherwise.



Orange cells are new reachability facts found by Warshall's algorithm (paths of length > 1).

Fig 10.4 — The adjacency matrix records direct edges only; the transitive closure records reachability by any path.

10.4.2 Warshall's Algorithm

Warshall's algorithm computes the transitive closure efficiently using three nested loops. Its central idea is to consider each vertex k in turn as a possible intermediate vertex on a path: if i can reach k and k can reach j , then i can reach j .

```
void warshall(int T[][N], int n) {
    /* T starts as a copy of the adjacency matrix */
    for (int k = 0; k < n; k++) /* intermediate vertex */
        for (int i = 0; i < n; i++) /* source vertex */
            for (int j = 0; j < n; j++) /* destination vertex */
                if (T[i][k] && T[k][j])
                    T[i][j] = 1;
}
```

The key rule applied at every step is simply:

$$T[i][j] = T[i][j] \text{ OR } (T[i][k] \text{ AND } T[k][j])$$

Note: The order of the loops matters: k must be the outermost loop. This ensures that when vertex k is considered, all paths using only vertices $0 \dots k-1$ as intermediates have already been found.

Example 10.1 — Warshall's algorithm

A directed graph on vertices 1, 2, 3, 4 has edges $1 \rightarrow 2$, $2 \rightarrow 3$ and $3 \rightarrow 4$. Find its transitive closure.

Solution

Direct edges give: $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 4$.

Using 2 as an intermediate: $1 \rightarrow 2$ and $2 \rightarrow 3 \Rightarrow 1 \rightarrow 3$.

Using 3 as an intermediate: $1 \rightarrow 3$ and $3 \rightarrow 4 \Rightarrow 1 \rightarrow 4$; $2 \rightarrow 3$ and $3 \rightarrow 4 \Rightarrow 2 \rightarrow 4$.

Closure: vertex 1 reaches 2, 3, 4; vertex 2 reaches 3, 4; vertex 3 reaches 4; vertex 4 reaches nothing.

Warshall's algorithm runs in $O(n^3)$ time and uses $O(n^2)$ space. The closely related Floyd-Warshall algorithm uses exactly the same triple loop but stores path costs instead of boolean reachability, computing the shortest distance between every pair of vertices.

bcalink.com

10.5 Graph Traversal and Spanning Forests

Graph traversal means visiting every vertex reachable from a starting vertex exactly once. Because a graph may contain cycles, every traversal algorithm must mark vertices as visited to avoid infinite looping — a requirement that does not arise when traversing a tree.

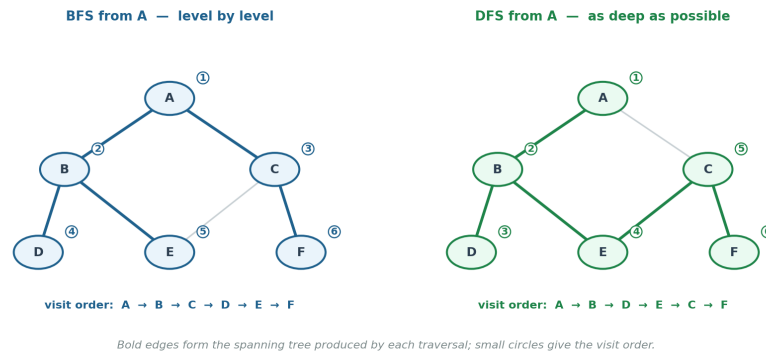


Fig 10.5 — Breadth-first and depth-first traversal of the same graph, each producing a different spanning tree.

10.5.1 Breadth-First Search (BFS)

BFS explores the graph level by level: it visits the start vertex, then all vertices at distance 1, then all at distance 2, and so on. It uses a queue to remember which vertices to visit next.

```
void BFS(int start) {
    bool visited[N] = { false };
    enqueue(start); visited[start] = true;
    while (!isEmpty()) {
        int v = dequeue();
        printf("%d ", v);
        for (each w adjacent to v)
            if (!visited[w]) {
                visited[w] = true;
                enqueue(w);
            }
    }
}
```

10.5.2 Depth-First Search (DFS)

DFS explores as deeply as possible along one branch before backtracking. It is naturally recursive, and its iterative form uses a stack in place of BFS's queue.

```

void DFS(int v, bool visited[]) {
    visited[v] = true;
    printf("%d ", v);
    for (each w adjacent to v)
        if (!visited[w])
            DFS(w, visited); /* recurse deeper */
}

```

Aspect	BFS	DFS
Data structure	Queue	Stack (or recursion)
Exploration order	Level by level (nearest first)	As deep as possible, then backtrack
Space complexity	$O(\text{width of the graph})$	$O(\text{depth of the graph})$
Time complexity	$O(n + e)$	$O(n + e)$
Typical uses	Shortest path in an unweighted graph, level-order processing	Cycle detection, topological sorting, connected components

10.5.3 Spanning Trees and Spanning Forests

A spanning tree of a connected graph G is a subgraph that includes every vertex of G , is connected, and contains no cycles. A spanning tree of a graph with n vertices always has exactly $n - 1$ edges.

Both BFS and DFS produce a spanning tree naturally: the edges actually used to reach previously unvisited vertices form the tree, and the remaining edges are discarded. BFS gives a breadth-first spanning tree and DFS a depth-first spanning tree — usually different trees for the same graph.

If the graph is disconnected, a single traversal cannot reach every vertex. The traversal is restarted from an unvisited vertex in each component, producing one spanning tree per component. This collection of spanning trees is called a spanning forest.

Note: A graph may have very many different spanning trees. When the edges carry weights, the one with the smallest total weight is the minimum spanning tree (MST), computed by algorithms such as Kruskal's.

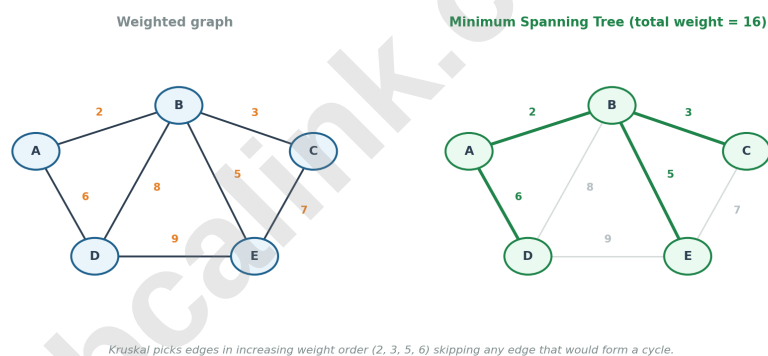
10.6 Minimum Spanning Trees: Kruskal's Algorithm

A minimum spanning tree of a weighted connected graph is a spanning tree whose total edge weight is as small as possible. MSTs answer practical questions such as: what is the cheapest set of cables that connects every building on a campus?

10.6.1 Kruskal's Algorithm

Kruskal's algorithm is greedy: it repeatedly takes the cheapest edge available that does not create a cycle.

- Step 1: Sort all edges of the graph in increasing order of weight.
- Step 2: Start with a forest in which every vertex is its own separate tree.
- Step 3: Take the next cheapest edge. If its two endpoints lie in different trees, add it to the MST and merge those trees. If they lie in the same tree, discard the edge — adding it would create a cycle.
- Step 4: Repeat Step 3 until the MST contains exactly $n - 1$ edges.



Kruskal picks edges in increasing weight order (2, 3, 5, 6) skipping any edge that would form a cycle.

Fig 10.6 — Kruskal's algorithm selects edges of weight 2, 3, 5 and 6 to build a minimum spanning tree of total weight 16.

The cycle test is performed efficiently using the Union-Find (disjoint-set) data structure, which can determine whether two vertices already belong to the same component in nearly constant time.

Step	Detail	Cost
Sorting the edges	Sort e edges by weight	$O(e \log e)$
Cycle detection	Union-Find operations across all edges	Nearly $O(e)$
Overall	Dominated by the sorting step	$O(e \log e)$

Example 10.2 — Kruskal's algorithm

A graph on vertices A, B, C, D, E has edges: A–B (2), B–C (3), B–E (5), A–D (6), C–E (7), B–D (8), D–E (9). Find the minimum spanning tree.

Solution

Sorted order: 2, 3, 5, 6, 7, 8, 9.

Take A–B (2) — no cycle, accept.

Take B–C (3) — no cycle, accept.

Take B–E (5) — no cycle, accept.

Take A–D (6) — no cycle, accept. We now have $4 = n - 1$ edges, so we stop.

MST edges: A–B, B–C, B–E, A–D — total weight = $2 + 3 + 5 + 6 = 16$.

10.6.2 Round-Robin Algorithm

The round-robin algorithm (also known as Borůvka's algorithm, and sometimes as Sollin's algorithm) is a third classical method for finding a minimum spanning tree. Rather than growing one tree at a time, it grows all components simultaneously in rounds.

- Step 1: Begin with each vertex as its own separate component.
- Step 2: In each round, every component independently selects the cheapest edge that connects it to a vertex outside itself.
- Step 3: Add all the selected edges to the spanning tree, merging the components they join.
- Step 4: Repeat until only a single component remains.

Because every component merges with at least one other in each round, the number of components at least halves per round. There are therefore at most $\log_2 n$ rounds, giving an overall complexity of $O(e \log n)$. Since the components choose their edges independently, the algorithm parallelises very well, which is why it is used in distributed and parallel computing.

Algorithm	Strategy	Complexity
Kruskal's	Greedily add the globally cheapest safe edge	$O(e \log e)$
Prim's	Grow one tree from a start vertex, always adding its cheapest outgoing edge	$O(e \log n)$
Round-Robin (Borůvka's)	Every component picks its cheapest outgoing edge each round	$O(e \log n)$

10.7 Shortest-Path Algorithms and the Greedy Approach

10.7.1 The Shortest-Path Problem

Given a weighted graph, the shortest-path problem asks for the path between two vertices whose total edge weight is minimum. Three variants are commonly distinguished:

- Single-source shortest path — find the shortest path from one source vertex to every other vertex (solved by Dijkstra's algorithm).
- Single-pair shortest path — find the shortest path between one specific pair of vertices.
- All-pairs shortest path — find the shortest path between every pair of vertices (solved by the Floyd-Warshall algorithm).

10.7.2 The Greedy Algorithm

A greedy algorithm builds a solution step by step, at each step choosing the option that looks best at that moment, without ever reconsidering an earlier choice. It never backtracks.

- Greedy choice property — a globally optimal solution can be reached by making a locally optimal choice at each step.
- Optimal substructure — an optimal solution to the problem contains optimal solutions to its subproblems.

When both properties hold, the greedy strategy is guaranteed to produce an optimal answer. Kruskal's algorithm, Prim's algorithm, Dijkstra's algorithm and Huffman coding are all greedy algorithms that provably find the optimum.

Note: Greedy does not always work. The classic counter-example is the 0/1 knapsack problem, where repeatedly taking the most valuable remaining item can give a worse result than the true optimum. Such problems require dynamic programming instead.

10.8 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-path problem for a weighted graph with non-negative edge weights. It maintains a tentative distance $d[v]$ for every vertex and greedily finalises them in increasing order of distance from the source.

- Step 1: Set $d[\text{source}] = 0$ and $d[v] = \infty$ for every other vertex. Mark all vertices as unvisited.
- Step 2: Among the unvisited vertices, select the one u with the smallest tentative distance $d[u]$. This is the greedy choice.
- Step 3: Mark u as visited; its distance $d[u]$ is now final and will never change again.

- Step 4 (relaxation): For every unvisited neighbour v of u , if $d[u] + \text{weight}(u, v) < d[v]$, then update $d[v] = d[u] + \text{weight}(u, v)$ and record u as v 's predecessor.
- Step 5: Repeat Steps 2–4 until every vertex has been visited.

Dijkstra's Algorithm — Shortest Paths from Source A

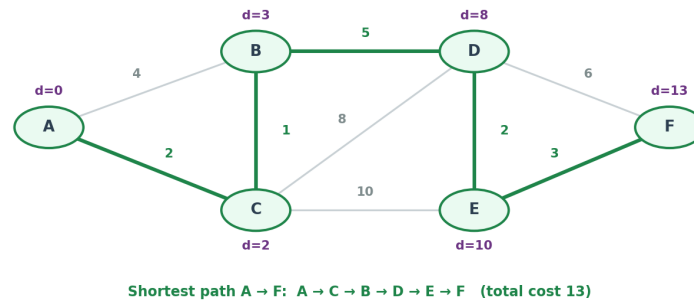


Fig 10.7 — Dijkstra's algorithm from source A; each vertex is labelled with its final shortest distance d .

```
void dijkstra(int graph[][N], int n, int src) {
    int dist[N]; bool visited[N] = { false };
    for (int i = 0; i < n; i++) dist[i] = INFINITY;
    dist[src] = 0;
    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(dist, visited, n); /* greedy choice */
        visited[u] = true; /* d[u] is final */
        for (int v = 0; v < n; v++) /* relaxation */
            if (!visited[v] && graph[u][v] &&
                dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
}
```

Example 10.3 — Dijkstra's algorithm

Using the graph of Fig 10.7, find the shortest distance from A to every other vertex. Edges: A–B (4), A–C (2), B–C (1), B–D (5), C–D (8), C–E (10), D–E (2), D–F (6), E–F (3).

Solution

Start: $d[A] = 0$, all others ∞ .

Visit A: relax neighbours $\rightarrow d[B] = 4$, $d[C] = 2$.

Smallest unvisited is C ($d = 2$). Visit C: via C, $d[B] = \min(4, 2+1) = 3$; $d[D] = 2+8 = 10$; $d[E] = 2+10 = 12$.

Smallest unvisited is B ($d = 3$). Visit B: $d[D] = \min(10, 3+5) = 8$.

Smallest unvisited is D ($d = 8$). Visit D: $d[E] = \min(12, 8+2) = 10$; $d[F] = 8+6 = 14$.

Smallest unvisited is E ($d = 10$). Visit E: $d[F] = \min(14, 10+3) = 13$.

Final distances: A = 0, B = 3, C = 2, D = 8, E = 10, F = 13.

Shortest path A $\rightarrow$ F is A $\rightarrow$ C $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E $\rightarrow$ F, of total cost 13.

Implementation	Time Complexity	Best suited for
Adjacency matrix with linear scan	$O(n^2)$	Dense graphs
Adjacency list with a binary min-heap	$O((n + e) \log n)$	Sparse graphs

Note: Dijkstra's algorithm fails if any edge has a negative weight, because once a vertex is marked visited its distance is assumed final — a later negative edge could otherwise still improve it. Graphs with negative weights require the Bellman-Ford algorithm instead.

Key Takeaways

Unit 10 — Key Takeaways

- ✓ A graph $G = (V, E)$ is a non-linear structure with no root and possible cycles; a tree is just a connected acyclic graph with $n - 1$ edges.
- ✓ Graphs are stored either as an adjacency matrix ($O(n^2)$ space, $O(1)$ edge lookup, good for dense graphs) or an adjacency list ($O(n + e)$ space, good for sparse graphs).
- ✓ The transitive closure records whether a path of any length exists between each pair of vertices; Warshall's algorithm computes it in $O(n^3)$ using $T[i][j] = T[i][j] \text{ OR } (T[i][k] \text{ AND } T[k][j])$.
- ✓ BFS uses a queue and explores level by level; DFS uses a stack or recursion and explores as deeply as possible. Both run in $O(n + e)$.
- ✓ The edges a traversal uses to reach new vertices form a spanning tree; for a disconnected graph, one tree per component gives a spanning forest.
- ✓ Kruskal's algorithm greedily adds the cheapest edge that creates no cycle, using Union-Find for the cycle test, and runs in $O(e \log e)$.
- ✓ The round-robin (Borůvka's) algorithm grows all components at once, each picking its cheapest outgoing edge per round, finishing in $O(e \log n)$ and parallelising well.
- ✓ A greedy algorithm makes the locally best choice at each step and never backtracks; it is optimal only when the greedy choice property and optimal substructure both hold.
- ✓ Dijkstra's algorithm greedily finalises the nearest unvisited vertex and relaxes its neighbours, finding single-source shortest paths — but only for non-negative edge weights.

— End of Unit 10 —