

Dynamic Programming

Introduction

Dynamic Programming (DP) is an important algorithm design technique used for solving complex optimization and decision-making problems. It is mainly used when a problem can be divided into smaller overlapping subproblems whose solutions can be reused.

The main idea behind Dynamic Programming is:

“Solve each subproblem only once and store its result for future use.”

Dynamic Programming was introduced by Richard Bellman in the 1950s.

DP is widely used in:

- Optimization problems
- Shortest path problems
- Resource allocation
- Scheduling
- Sequence analysis

Definition

Dynamic Programming is a method of solving problems by:

1. breaking them into smaller subproblems,
2. solving each subproblem once,
3. storing the solutions,
4. and reusing them whenever required.

Basic Idea of Dynamic Programming

In many recursive problems, the same subproblems are solved repeatedly.

Dynamic Programming avoids this repetition by:

- storing previously computed values,
- reducing unnecessary calculations,
- improving efficiency.

Example: Fibonacci Series

The Fibonacci recurrence relation is:

$$F(n) = F(n - 1) + F(n - 2)$$

Recursive Computation Problem

To calculate:

$$F(5)$$

the recursive method repeatedly computes:

$$F(3), F(2), F(1)$$

multiple times.

This causes:

- redundant computations,
- high time complexity.

Dynamic Programming Solution

Store already computed Fibonacci values in an array.

Thus:

- each value is calculated only once,
- time complexity reduces significantly.

Characteristics of Dynamic Programming

1. Overlapping Subproblems: The same subproblems appear repeatedly during computation.

Example

While calculating:

$$F(5)$$

the value:

$$F(3)$$

is computed multiple times.

DP stores such values to avoid recomputation.

2. Optimal Substructure: The optimal solution of a problem can be constructed from optimal solutions of its smaller subproblems.

Example

Shortest path problems.

If the shortest path from:

$$A \rightarrow C$$

passes through:

$$B$$

then:

$$A \rightarrow B$$

and

$$B \rightarrow C$$

must also be shortest paths.

Steps in Dynamic Programming

Step 1: Identify Subproblems

Break the main problem into smaller problems.

Step 2: Form Recurrence Relation

Express solution recursively.

Example

$$F(n) = F(n - 1) + F(n - 2)$$

Step 3: Solve Subproblems

Compute smaller subproblems first.

Step 4: Store Results

Save computed values in a table or array.

Step 5: Construct Final Solution

Use stored values to build final answer.

Approaches in Dynamic Programming

1. Top-Down Approach (Memoization)

Concept

- Uses recursion.
- Stores computed values.
- If value already exists, reuse it.

Working

1. Solve recursively.
2. Before solving, check whether answer already exists.
3. Return stored value if available.

Advantages

- Easy to implement.
- Solves only required subproblems.

Disadvantages

- Recursive overhead.
- Uses stack memory.

Memoization Example

```
Fib(n)
{
    if  $n \leq 1$ 
        return  $n$ 

    if value already stored
        return stored value

    store  $Fib(n-1) + Fib(n-2)$ 

    return stored value
}
```

2. Bottom-Up Approach (Tabulation)**Concept**

- Solve smallest subproblems first.
- Build solution iteratively.

Working

1. Create table.
2. Fill table from base cases upward.
3. Final value gives solution.

Advantages

- Faster execution.
- No recursion overhead.

Disadvantages

- Sometimes computes unnecessary subproblems.

Tabulation Example

```

Fib(n)
{
    F[0] = 0
    F[1] = 1

    for i = 2 to n
    {
        F[i] = F[i-1] + F[i-2]
    }

    return F[n]
}

```

Difference Between Recursion and Dynamic Programming

Recursion	Dynamic Programming
Repeats subproblems	Solves once and stores
Higher time complexity	Lower time complexity
No storage	Uses memory/table
Simpler logic	More optimized

Difference Between Greedy Method and Dynamic Programming

Greedy Method	Dynamic Programming
Makes local optimal choice	Solves all subproblems
Faster	More computation
May fail to give optimal solution	Guarantees optimal solution
Less memory	More memory
Example: Prim's Algorithm	Example: 0/1 Knapsack

Applications of Dynamic Programming**1. 0/1 Knapsack Problem**

Objective

Maximize profit within knapsack capacity.

Items cannot be divided.

2. Matrix Chain Multiplication**Objective**

Find optimal order of matrix multiplication to minimize computations.

3. Longest Common Subsequence (LCS)**Objective**

Find longest common subsequence between two strings.

Applications

- DNA matching
- Text comparison
- Version control systems

4. Floyd Warshall Algorithm**Objective**

Find shortest paths between all pairs of vertices.

5. Bellman Ford Algorithm**Objective**

Find shortest paths even with negative edge weights.

6. Travelling Salesman Problem (TSP)**Objective**

Find shortest route visiting all cities exactly once.

7. Optimal Binary Search Tree**Objective**

Minimize average search cost.

Time Complexity Improvement

Dynamic Programming often converts:

- exponential-time algorithms
into
- polynomial-time algorithms.

Example**Fibonacci Recursive Complexity**

$$O(2^n)$$

DP Complexity

$$O(n)$$

Advantages of Dynamic Programming

- Avoids Repeated Computation: Stores results for reuse.
- Produces Optimal Solutions: Works well for optimization problems.
- Improves Efficiency: Reduces time complexity significantly.
- Handles Complex Problems: Suitable for difficult combinational problems.

Limitations of Dynamic Programming

- High Memory Usage: Requires storage tables.
- Difficult Design: Finding recurrence relation can be challenging.
- Not Suitable for All Problems: Requires overlapping subproblems and optimal substructure.
- Increased Implementation Complexity: Compared to simple greedy algorithms.

Real-Life Applications

- Networking: Shortest path routing.
- Bioinformatics: DNA sequence alignment.
- Artificial Intelligence: Decision-making and planning.
- Finance: Investment optimization.
- Operations Research: Resource allocation and scheduling.

Multistage Graph

A multistage graph is a special type of directed weighted graph in which the vertices are divided into different stages. The edges in the graph connect vertices from one stage to the next stage only. Since the edges always move in the forward direction, the graph does not contain cycles and is therefore a Directed Acyclic Graph (DAG).

Multistage graphs are widely used in Dynamic Programming because many optimization problems can be represented in stages. The main objective is generally to determine the minimum cost path from a source vertex to a destination vertex.

A multistage graph typically contains:

- a source vertex in the first stage,
- intermediate vertices in the middle stages,
- and a destination vertex in the final stage.

Each edge in the graph has an associated cost or weight. The total cost of a path is obtained by adding the weights of all edges present in that path. Among all possible paths from source to destination, the path having minimum total cost is considered the optimal path.

The important property of a multistage graph is that the solution to a problem at one stage depends on the solutions of the subsequent stages. This property makes Dynamic Programming an efficient technique for solving multistage graph problems.

Suppose:

$$Cost(v)$$

represents the minimum cost from a vertex v to the destination. Then the Dynamic Programming recurrence relation is:

$$Cost(v) = \min [w(v, u) + Cost(u)]$$

where:

- $w(v, u)$ is the cost of the edge from vertex v to vertex u ,
- and u is a vertex in the next stage.

Algorithm Steps**Step 1**

Assign:

$$\text{Cost}(\text{destination}) = 0$$

Step 2

Move backward stage by stage.

Step 3

For each vertex:

- calculate cost of all outgoing edges,
- select minimum value.

Step 4

Store computed costs.

Step 5

Trace path from source using stored decisions.

Pseudo Code

```
MultistageGraph(G)
{
    cost[destination] = 0

    for each stage from last-1 to first
    {
        for each vertex v in stage
        {
            cost[v] = infinity

            for each adjacent vertex u
            {
                if w(v,u) + cost[u] < cost[v]
                {
                    cost[v] = w(v,u) + cost[u]

                    d[v] = u
                }
            }
        }
    }

    return cost[source]
}
```


Time Complexity

If:

- V = number of vertices
- E = number of edges

Then complexity is:

$$O(E)$$

because each edge is processed once.

Space Complexity

$$O(V)$$

for storing:

- costs,
- decisions,
- path information.

Advantages of Multistage Graph

- **Efficient Optimization:** Finds optimal solution systematically.
- **Avoids Repeated Computation:** Uses Dynamic Programming storage
- **Suitable for Sequential Problems:** Best for stage-wise decision problems.
- **Faster Computation:** Processes each edge only once.

Limitations

- **Applicable Mainly to DAGs:** Cycles complicate computation.
- **Requires Stage Division:** Vertices must be clearly partitioned.
- **Large Graph Complexity:** Very large graphs require more memory

Applications of Multistage Graph

- **Shortest Path Problems:** Network routing optimization.
- **Resource Allocation:** Optimal distribution of resources across stages.
- **Production Planning:** Manufacturing process optimization.
- **Transportation Systems:** Minimum travel cost calculation.
- **Decision Making Systems:** Sequential optimization problems.
- **Communication Networks:** Efficient packet routing.

Real-Life Example

Suppose a traveller moves through multiple cities to reach a destination.

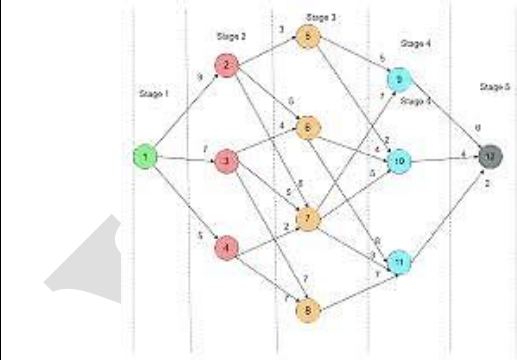
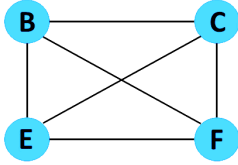
Each stage represents:

- a group of cities,
- and edges represent travel costs.

Dynamic Programming determines the cheapest path between source and destination.

YBKR

Difference Between Multistage Graph and General Graph

Multistage Graph	General Graph
A multistage graph is divided into different stages or levels.	A general graph does not have any stage-wise structure.
Edges move only from one stage to the next stage in forward direction.	Edges can connect any pair of vertices in any direction.
It is usually a Directed Acyclic Graph (DAG) and does not contain cycles.	A general graph may contain cycles.
Mainly used for Dynamic Programming and optimization problems.	Used for general graph problems such as traversal, connectivity, and routing.
Finding shortest or optimal paths is easier due to stage-wise arrangement.	Problems may become more complex because of unrestricted connections.
Dynamic Programming techniques are commonly applied efficiently.	Various algorithms like DFS, BFS, Dijkstra's, etc., are used depending on the problem.
Every path generally starts from a source stage and ends at a destination stage.	No fixed source or destination stages are necessary.
Time complexity is usually lower because computation proceeds stage by stage.	Complexity may be higher due to cycles and unrestricted traversal.
	

All Pairs Shortest Path Problem (APSP)

The **All Pairs Shortest Path Problem** is an important graph optimization problem in Dynamic Programming and graph theory. The objective of this problem is to find the shortest paths between every pair of vertices in a weighted graph.

Unlike the Single Source Shortest Path problem, where the shortest path is determined from one source vertex to all other vertices, the All Pairs Shortest Path problem computes shortest paths between all possible pairs of vertices in the graph.

The most commonly used algorithm for solving this problem is the **Floyd-Warshall Algorithm**, which is based on Dynamic Programming.

Definition

The All Pairs Shortest Path Problem is the problem of finding the minimum distance between every pair of vertices in a weighted graph.

If a graph contains:

V

vertices, then shortest paths are computed for all:

$V \times V$

pairs of vertices.

Need for All Pairs Shortest Path

The APSP problem is useful in situations where:

- shortest distances between all locations are required,
- routing information must be maintained,
- and optimal communication paths are needed.

Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm is a Dynamic Programming algorithm used to compute shortest paths between all pairs of vertices in a weighted graph.

It works for:

- directed graphs,
- undirected graphs,
- graphs with positive or negative edge weights.

However, it does not work correctly for graphs containing negative weight cycles.

Basic Idea

The algorithm checks whether passing through an intermediate vertex gives a shorter path between two vertices.

Suppose:

- i = source vertex

- j = destination vertex
- k = intermediate vertex

Then the shortest path is updated using:

$$d[i][j] = \min (d[i][j], d[i][k] + d[k][j])$$

This means:

- compare the current shortest distance,
- with the distance obtained through vertex k ,
- and choose the smaller value.

Dynamic Programming Principle

The problem satisfies:

- **Optimal Substructure**
- **Overlapping Subproblems**

The shortest path between two vertices can be constructed from shorter paths involving intermediate vertices.

Algorithm Steps

1. Create a distance matrix from graph weights.
2. Assign:
 - 0 for same vertices,
 - infinity where no edge exists.
3. Consider each vertex as an intermediate vertex.
4. Update shortest distances using Dynamic Programming relation.
5. Continue until all vertices are processed.

Pseudo Code

```
FloydWarshall(G)
{
    initialize distance matrix D

    for k = 1 to n
    {
        for i = 1 to n
        {
            for j = 1 to n
            {
                D[i][j] = min(D[i][j],
                             D[i][k] + D[k][j])
            }
        }
    }

    return D
}
```

Time Complexity

The Floyd–Warshall Algorithm uses three nested loops.

Therefore:

$$O(V^3)$$

Where:

- V = number of vertices.

Space Complexity

$$O(V^2)$$

because a distance matrix is stored.

Advantages

- Computes shortest paths between all pairs simultaneously.
- Simple and systematic algorithm.
- Works with negative edge weights.
- Dynamic Programming guarantees optimal solution.

Limitations

- High time complexity for large graphs.
- Requires large memory for distance matrix.
- Cannot handle negative weight cycles.

Applications

- Network Routing: Finding shortest communication paths.
- Transportation Systems: Determining minimum travel distances.
- Airline and Railway Networks: Optimal route calculation.
- Social Networks: Finding minimum connection distances between users.
- GPS and Navigation Systems: Computing shortest routes between all locations.

Difference Between Single Source Shortest Path and All Pairs Shortest Path

Single Source Shortest Path (SSSP)	All Pairs Shortest Path (APSP)
Finds the shortest path from one source vertex to all other vertices in the graph.	Finds the shortest paths between every pair of vertices in the graph.
Only one vertex is considered as the source.	Every vertex is considered as both source and destination.
Algorithms such as Dijkstra's and Bellman-Ford are commonly used.	Floyd–Warshall Algorithm is commonly used.
Requires less computation compared to APSP.	Requires more computation because all vertex pairs are processed.
Time complexity is generally lower.	Time complexity is usually higher, commonly $O(V^3)$.
Suitable when shortest paths are needed from only one location.	Suitable when shortest distances between all locations are required.

Single Source Shortest Path (SSSP)	All Pairs Shortest Path (APSP)
Uses less memory.	Requires more memory to store distance matrices.
Example: Finding shortest path from one city to all other cities.	Example: Finding shortest distances between every pair of cities in a map.

YBKR

Travelling Salesperson Problem

The Travelling Salesperson Problem (TSP) is one of the most important optimization problems in computer science and graph theory. The problem involves a salesperson who must visit a number of cities exactly once and finally return to the starting city. The objective is to find the route with the minimum total travelling cost or distance.

In the Travelling Salesperson Problem, each city is represented as a vertex and the distance between cities is represented as weighted edges in a graph. The solution to the problem is a tour that visits every vertex exactly one time and returns to the source vertex. Such a tour is called a Hamiltonian Cycle.

The main goal of TSP is to minimize the total travel cost. If there are n cities, there can be many possible tours. Among all possible tours, the tour having the minimum total distance is considered the optimal solution.

The Travelling Salesperson Problem is the problem of finding the shortest possible route that:

1. visits every city exactly once,
2. returns to the starting city,
3. and minimizes the total travel cost.

Problem Statement

Given:

- a set of cities,
- distances between every pair of cities,

find the minimum cost Hamiltonian cycle.

Important Terms

1. Tour

A complete path visiting all cities exactly once and returning to the source.

2. Hamiltonian Cycle

A cycle that visits every vertex exactly once and returns to the starting vertex.

3. Cost Matrix

A matrix representing distances between cities.

Example

Suppose there are four cities:

A, B, C, D

Distance matrix:

0	10	15	20
10	0	35	25
15	35	0	30
20	25	30	0

The objective is to determine the minimum cost tour.

Dynamic Programming Approach

Dynamic Programming solves TSP by:

- dividing the problem into smaller subproblems,
- storing intermediate solutions,
- avoiding repeated computations.

Principle of Dynamic Programming

The problem satisfies:

1. **Optimal Substructure**
2. **Overlapping Subproblems**

The optimal tour can be constructed from optimal tours of smaller subsets of cities.

State Representation

Suppose:

- i = current city
- S = set of remaining cities

Then:

$$Cost(i, S)$$

represents the minimum cost required to:

- start from city i ,
- visit all cities in set S ,
- return to source.

Recurrence Relation

The Dynamic Programming relation is:

$$Cost(i, S) = \min_{j \in S} [d(i, j) + Cost(j, S - \{j\})]$$

Where:

- $d(i, j)$ = distance from city i to city j ,
- $S - \{j\}$ = remaining cities after visiting j .

Base Condition

When no cities remain:

$$Cost(i, \emptyset) = d(i, source)$$

This means:

- directly return to starting city.

Algorithm Steps

1. Start from source city.
2. Generate subsets of cities.
3. Compute minimum cost recursively.
4. Store computed results.
5. Reuse stored values.
6. Determine minimum tour cost.

Pseudo Code

```
TSP(i, S)
{
    if S is empty
        return distance(i, source)

    minimum = infinity

    for each city j in S
    {
        cost = distance(i,j) +
              TSP(j, S-{j})

        minimum = min(minimum, cost)
    }

    return minimum
}
```

Time Complexity**Brute Force Method**

$$O(n!)$$

because all possible tours are checked.

Dynamic Programming Method

$$O(n^2 2^n)$$

which is much better than factorial complexity.

Space Complexity

$$O(n 2^n)$$

because DP tables store subsets and states.

Advantages of Dynamic Programming in TSP

- Reduces repeated computations.
- Faster than brute force method.
- Produces optimal solution.
- Efficient for moderate-sized problems.

Limitations

- High memory usage.
- Complexity still large for very big graphs.
- Difficult implementation.

Thus, the Travelling Salesperson Problem is an important optimization problem that aims to determine the minimum cost tour covering all cities exactly once and returning to the source city. Dynamic Programming provides an efficient technique for solving this problem compared to exhaustive search methods.

YBKR

YBKR