# Zkouška EN ústní

## ▼ Introduction to DP

### 1. Core Concepts

| Term | Definition |
|---|---|
| **Dynamic Programming** | Method for solving multi-stage decision problems by breaking them into smaller subproblems. Relies on Bellman's **Principle of Optimality**. |
| **Stage** | Discrete time index $k = 0, 1, \ldots, N$. Decisions are made stage by stage. |
| **State xkx_kxk** | Information summarizing the past that is relevant for future optimization. |
| **Control** $u_k$ | Decision chosen at stage $k$. Must satisfy $u_k \in U_k(x_k)$. |
| **Disturbance** $w_k$ | Random variable affecting transitions. Distribution depends only on current state & control (**Markov property**). |
| **System Dynamics** | $x_{k+1} = f_k(x_k, u_k, w_k)$ |
| **Cost Function** | Additive form: terminal cost $g_N(x_N)$ + stage costs $g_k(x_k, u_k, w_k)$. Optimization based on expected total cost. |
| **Policy** $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ | Set of functions mapping states to decisions: $u_k = \mu_k(x_k)$. |
| **Admissible Policy** | Policy where every $\mu_k(x_k) \in U_k(x_k)$. |
| **Optimal Policy** $\pi^*$ | Minimizes expected total cost. |
| **Optimal Cost / Value Function** $J^*(x_0)$ | Minimum achievable expected cost from initial state $x_0$. |

### 2. Open-Loop vs. Closed-Loop

| Mode | Description | Pros/Cons |
|---|---|---|
| **Open-loop** | Choose all controls $u_0, \ldots, u_{N-1}$ at time 0. No adaptation to disturbances. | Simpler but suboptimal—cannot react to new information. |

| Mode | Description | Pros/Cons |
|---|---|---|
| **Closed-loop** | Choose $u_k$ at time $k$ based on current state $x_k$. | Always ≥ performance; yields **value of information**. |

## 3. Transition Models

### Continuous / General Form

- $x_{k+1} = f_k(x_k, u_k, w_k)$

### Discrete-State Transition Probabilities

- $p_{ij}(u, k) = P\{x_{k+1} = j \mid x_k = i, u_k = u\}$

  Equivalent to describing dynamics via random disturbance $w_k$.

## 4. Expected Total Cost Under Policy

$J_\pi(x_0) = \mathbb{E}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right]$

Optimal cost:

$J^*(x_0) = \min_\pi J_\pi(x_0)$

## 5. Principle of Optimality (Bellman)

> Given an optimal policy, its tail starting from any reachable state at time $i$ is itself optimal for the subproblem from $i$ to $N$.

This enables **backward induction** → foundation of the DP algorithm.

## 6. DP Backward Recursion (Finite Horizon)

From slide proposition:

1. **Initialization (terminal stage)**

   $J_N(x_N) = g_N(x_N)$

2. **Backward recursion for $k = N-1, \ldots, 0$**

   $J_k(x_k) = \min_{u_k \in U_k(x_k)} \mathbb{E}_{w_k}\left[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))\right]$

3. **Optimal control law**

$$\mu_k^*(x_k) = \arg\min_{u_k \in U_k(x_k)} \mathbb{E}_{w_k}[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))]$$

## 7. Interpretation of $J_k(x_k)$

- Cost-to-go function from state $x_k$ at time $k$.
- Represents optimal expected cost for a remaining horizon of $N - k$.
- Slide note: Computational effort grows with number of states → **curse of dimensionality**.

## 8. Example – Inventory Control (Stochastic)

### Variables

| Symbol | Description |
|---|---|
| $x_k$ | Stock at start of period $k$. |
| $u_k$ | Units ordered at start of period $k$. |
| $w_k$ | Random demand in period $k$. |

### Dynamics

$x_{k+1} = x_k + u_k - w_k$

### Cost Components

- Holding / shortage cost: $r(x_k)$
- Ordering cost: $cu_k$
- Terminal cost: $R(x_N)$

### Objective

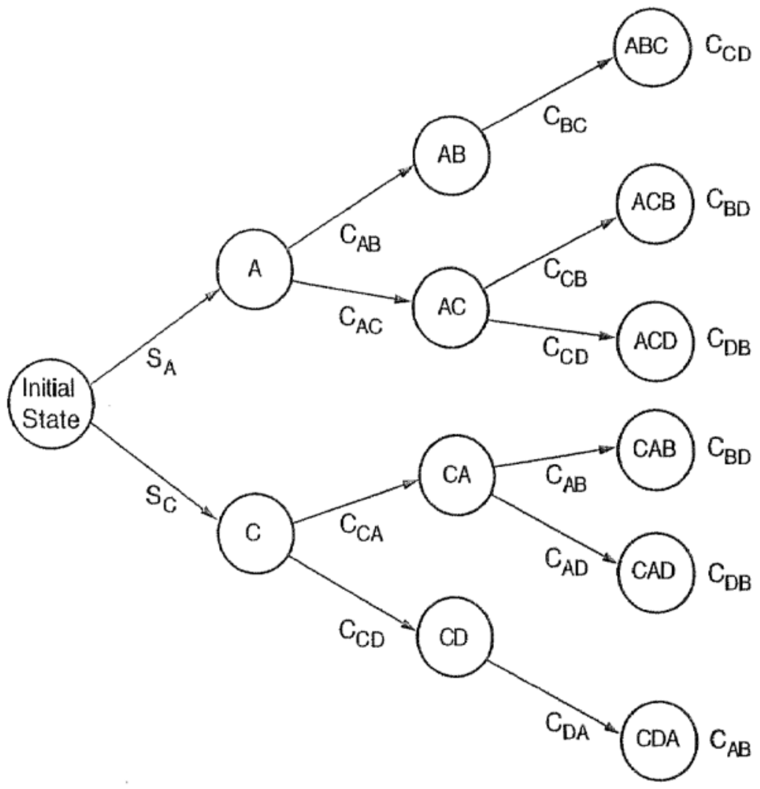
$\mathbb{E}\left[R(x_N) + \sum_{k=0}^{N-1}(r(x_k) + cu_k)\right]$

DP chooses ordering rule $u_k = \mu_k(x_k)$.

## 9. Example – Deterministic Scheduling

- Sequence operations A,B,C,D with precedence constraints.

- Each decision chooses next operation; cost is setup cost + startup cost.
- Represented as **transition graph**.



| | |
|---|---|
| $S_A$ | 5 |
| $S_C$ | 3 |
| $C_{AB}$ | 2 |
| $C_{AC}$ | 3 |
| $C_{AD}$ | 4 |
| $C_{BC}$ | 3 |
| $C_{BD}$ | 1 |
| $C_{CA}$ | 4 |
| $C_{CB}$ | 4 |
| $C_{CD}$ | 6 |
| $C_{DA}$ | 3 |
| $C_{DB}$ | 3 |

- Solved naturally via DP as a finite-state deterministic problem.

## 10. Why DP Works

- Converts a full-horizon optimization into nested smaller problems.
- Backward induction computes optimal decisions *stage by stage*.
- Optimal policy is *state-feedback* → inherently adaptive.

# Key Formulas Summary

| Concept | Formula |
|---|---|
| **Dynamics** | $x_{k+1} = f_k(x_k, u_k, w_k)$ |
| **Total Expected Cost** | $J_\pi(x_0)$ as above |

| Concept | Formula |
|---|---|
| **DP Recursion** | $J_k(x_k) = \min_u \mathbb{E}[g_k + J_{k+1}]$ |
| **Optimal Control Law** | $\mu_k^*(x_k) = \arg\min_u \mathbb{E}[g_k + J_{k+1}]$ |
| **Terminal Condition** | $J_N(x_N) = g_N(x_N)$ |

## ▼ Minimax

## 1. Core Idea of Minimax Control

Minimax (robust) control addresses decision-making under uncertainty **when no probabilistic description is available**.

Instead of assuming known probability distributions for disturbances, we only know **sets** within which disturbances may lie.

The goal is to design a policy that performs well **under the worst possible disturbance sequence**.

Formally, the objective is:

$$\min_{\pi\in\Pi} \max_{w\in W} J(\pi, w)$$

This "worst-case" approach ensures **robustness** in situations where uncertainty cannot be modeled stochastically.

## 2. Problem Structure

Disturbances are assumed to satisfy:

$$w_k \in W_k(x_k, u_k)$$

A policy is a sequence of state-feedback functions:

$$\pi = \{\mu_0, \mu_1, \dots, \mu_{N-1}\}$$

The worst-case cost associated with a policy is:

$$J_\pi(x_0) = \max_{w_k\in W_k(x_k,\mu_k(x_k))} \left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right]$$

This replaces the **expected cost** from stochastic DP with a **maximum cost** over all admissible disturbances.

## 3. Minimax Dynamic Programming Algorithm

Just like in classical DP, the minimax algorithm proceeds backward.

The difference is that the expectation operator is replaced by a maximization over disturbances.

### Terminal stage

$J_N(x_N) = g_N(x_N)$

### Backward recursion

$J_k(x_k) = \min_{u_k \in U_k(x_k)} \max_{w_k \in W_k(x_k,u_k)} [g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))]$

The controller selects the action that gives the **best performance under the worst disturbance**.

## 4. State Augmentation (Yellow Highlights)

When assumptions of the classical DP model are violated (e.g., dependence on earlier states or controls), we can reformulate the problem through **state augmentation**.

General rule:

> Include in the state all information available at time kkk that is relevant for choosing $u_k$.

Examples:

### Time lags

If dynamics depend on earlier values:

$x_{k+1} = f_k(x_k, x_{k-1}, u_k, u_{k-1}, w_k)$

Introduce:

- $y_k = x_{k-1}$
- $s_k = u_{k-1}$

New augmented state:

$\tilde{x}_k = (x_k, y_k, s_k)$

**Consequence:** State space expands dramatically.

If original system had nnn states and mmm controls, augmented state may have up to:

$$n \times n \times m$$

## 5. Correlated Disturbances

If disturbances follow a correlation model:

$$w_k = \lambda w_{k-1} + \xi_k$$

Introduce an additional state variable:

$$y_k = w_{k-1}$$

New system state:

$$\tilde{x}_k = (x_k, y_k)$$

This converts correlated noise into a DP-compatible form.

## 6. Simplification When Part of the State Is Uncontrollable

If the state consists of $(x_k, y_k)$ but only $x_k$ can be influenced by control, computation can be reduced.

Define the reduced cost-to-go:

$$\hat{J}_k(x_k) = E_{y_k}\{J_k(x_k, y_k) \mid x_k\}$$

This allows DP to operate over the **controllable** component only, while still yielding an optimal control law defined over the full state.

## 7. Parking Problem (Illustrative Example)

In the parking problem, each space is either **free** or **taken**, and the driver must decide whether to park or move on.

The state distinguishes:

- $(k, F)$: space $k$ is free
- $(k, \bar{F})$: space $k$ is taken
- Terminal state $t$
- Garage with cost $C$

The cost-to-go functions are:

$$J_k^*(F) = \begin{cases} \min[c(k),\, p(k+1)J_{k+1}^*(F) + (1-p(k+1))J_{k+1}^*(\bar{F})], & k < N-1 \\ \min[c(N-1), C], & k = N-1 \end{cases}$$

$$J_k^*(\bar{F}) = \begin{cases} p(k+1)J_{k+1}^*(F) + (1-p(k+1))J_{k+1}^*(\bar{F}), & k < N-1 \\ C, & k = N-1 \end{cases}$$

Define the expected cost before knowing whether a space is free:

$$\hat{J}_k = p(k)\min[c(k), \hat{J}_{k+1}] + (1-p(k))\hat{J}_{k+1}$$

### Optimal policy :

> Park at space $k$ if it is free and $c(k) \le \hat{J}_{k+1}$.

## Key Takeaways (What to Say on an Exam)

- Minimax control is used when uncertainty **cannot be described probabilistically**.
- Disturbances are assumed to lie in **known sets**, and the objective is to minimize the **worst-case** cost.
- The dynamic programming recursion becomes a **min–max** problem instead of **min–expectation**.
- State augmentation is used to handle time lags or correlated disturbances.
- When some state components are uncontrollable, DP can be simplified by averaging over them.
- The parking problem illustrates DP with uncontrollable randomness and reduced-state formulation.

## ▼ Determinic finite-state problems

## 1. Deterministic Systems: Key Characteristics

A deterministic system is one where each disturbance $w_k$ can take **only one value**.

Such problems occur naturally when uncertainty is negligible or when a stochastic model is approximated by fixing disturbances to a typical value.

A fundamental property of deterministic systems is:

> Feedback provides no cost advantage.
>
> Minimizing over feedback policies$\{\mu_0, \ldots, \mu_{N-1}\}$ yields the same optimal cost as minimizing over open control sequences $\{u_0, \ldots, u_{N-1}\}$, because future states evolve **predictably** under deterministic dynamics.

Given a policy, the state evolution is fully known:

$$x_{k+1} = f_k(x_k, \mu_k(x_k)), \qquad u_k = \mu_k(x_k)$$

This distinguishes deterministic problems from stochastic ones and has computational consequences:

deterministic continuous-space problems can be solved by variational or gradient-based optimization, whereas DP remains useful when constraints or discrete structure are present.

## 2. Deterministic Finite-State Systems and the Shortest Path Interpretation

Consider a deterministic problem where each stage kkk has a **finite state set** SkS_kSk.

Every control uku_kuk induces a transition:

$$x_{k+1} = f_k(x_k, u_k)$$

with associated cost $g_k(x_k, u_k)$.

This leads to a natural **graph representation**:

- **Nodes** represent states at each stage.
- **Arcs** represent feasible transitions $x_k \to x_{k+1}$.
- **Arc cost** = transition cost $g_k(x_k, u_k)$.
- An artificial terminal node $t$ is added, and each final state $x_N$ has an arc $(x_N \to t)$ with cost $g_N(x_N)$.

## Core Interpretation

> A deterministic finite-state problem is equivalent to finding a minimum-length (shortest) path from the initial node $s$ to the terminal node $t$.

A **path** is a sequence of arcs

$(j_1, j_2), (j_2, j_3), \dots, (j_{k-1}, j_k)$

and its **length** is the sum of arc costs.

This graphical formulation makes DP identical to a shortest-path computation.

---

## 3. DP Algorithm for Deterministic Finite-State Problems

Using the notation:

- $a_{i,j}^k$: cost of transition at stage $k$ from state$i \in S_k$ to state $j \in S_{k+1}$
- $N = gN(i)$ = g_N(i)ai,tN=gN(i): terminal cost
- ai,jk=∞a_{i,j}^k = \inftyai,jk=∞ if no control allows the transition

Dynamic programming becomes:

### Terminal stage

JN(i)=ai,tN,i∈SNJ_N(i) = a_{i,t}^N, \qquad i \in S_N

JN(i)=ai,tN,i∈SN

### Backward recursion

Jk(i)=minj∈Sk+1[ai,jk+Jk+1(j)],k=0,…,N−1J_k(i)
=
\min_{j \in S_{k+1}} \left[ a_{i,j}^k + J_{k+1}(j) \right],
\qquad k = 0,\dots,N-1

Jk(i)=j∈Sk+1min[ai,jk+Jk+1(j)],k=0,…,N−1

The optimal cost:

J0(s)J_0(s)

J0(s)

is exactly the **length of the shortest path from** sss **to** ttt.

## 4. Forward DP Algorithm (Special Case)

Although DP is usually backward, deterministic finite-state problems allow an equivalent **forward algorithm**.

Key observation (yellow highlight):

> An optimal path from sss to ttt is also optimal in a reverse graph, where all arcs are reversed but their lengths remain unchanged.

The forward DP computes **cost-to-arrive** from sss:

J~N(j)=as,j0,j∈S1\tilde{J}_N(j) = a_{s,j}^0, \quad j \in S_1

J~N(j)=as,j0,j∈S1

J~k(j)=mini∈SN−k[ai,jN−k+J~k+1(i)]\tilde{J}_k(j) = \min_{i \in S_{N-k}} \left[a_{i,j}^{N-k} + \tilde{J}_{k+1}(i)\right]

J~k(j)=i∈SN−kmin[ai,jN−k+J~k+1(i)]

The final cost satisfies:

J~0(t)=J0(s)\tilde{J}_0(t) = J_0(s)

J~0(t)=J0(s)

Forward DP is useful in **real-time applications** where stage-kkk data becomes available only when stage kkk is reached.

## 5. Shortest Path Problems as Deterministic DP

Any classical shortest-path problem can be formulated as a deterministic finite-state DP problem.

Given:

- nodes {1,2,…,N,t}\{1,2,\dots,N,t\}{1,2,…,N,t},
- arc cost aija_{ij}aij,
- aij=∞a_{ij} = \inftyaij=∞ if arc does not exist,

Goal: find shortest path from each node iii to ttt.

Assumption (yellow highlight):

> No cycle may have negative total length, ensuring optimal paths never need more than NNN moves.

Hence we can require exactly NNN moves by allowing “degenerate moves” i→ii \to ii→i with zero cost.

DP definition:

Jk(i)=optimal cost of reaching t in N−k movesJ_k(i) = \text{optimal cost of reaching }t\text{ in }N-k\text{ moves}

Jk(i)=optimal cost of reaching t in N−k moves

DP equations:

Jk(i)=minj=1,…,N[aij+Jk+1(j)],k=0,…,N−2J_k(i) = \min_{j=1,\dots,N} [a_{ij} + J_{k+1}(j)], \quad k=0,\dots,N-2

Jk(i)=j=1,…,Nmin[aij+Jk+1(j)],k=0,…,N−2

JN−1(i)=ai,tJ_{N-1}(i) = a_{i,t}

JN−1(i)=ai,t

If degenerate moves appear, it simply means the actual shortest path uses fewer than NNN transitions.

---

## Key Takeaways (What to Say on the Exam)

- Deterministic systems have **predictable** state evolution; feedback offers **no advantage**.
- Finite-state deterministic DP problems map naturally to **shortest-path problems**.
- DP recursions compute shortest paths via backward or forward algorithms.
- Graph representation: **states = nodes**, **controls = arcs**, **transition cost = arc length**.
- DP and shortest-path formulations are fully equivalent.
- Forward DP is useful when information is revealed sequentially.
- Any shortest path problem can be converted into deterministic DP under the assumption of **no negative cycles**.

## ▼ Shortest path methods