

ДОДАТОК А

Код парсингу силабусів

```

import re
import json
from pathlib import Path
from dataclasses import dataclass, field, asdict
from typing import Optional

import pdfplumber

@dataclass
class Topic:
    number: int
    title: str
    description: str

@dataclass
class Module:
    number: int
    title: str
    topics: list[Topic] = field(default_factory=list)

@dataclass
class Competency:
    code: str
    text: str

@dataclass
class Course:
    file: str
    code: Optional[str] = None
    name: str = ""
    specialization: str = ""
    level: str = ""
    credits: Optional[float] = None
    semester: Optional[str] = None
    prerequisites_raw: str = ""
    prerequisites: list[str] = field(default_factory=list)
    is_school_only: bool = False
    annotation_full: str = ""
    modules: list[Module] = field(default_factory=list)
    competencies: list[Competency] = field(default_factory=list)
    raw_text: str = field(default="", repr=False)

NAME_OVERRIDES = {
    "cs.5655-kvalifikaczijna-robota.pdf": "Кваліфікаційна робота",
    "cs.5878-praktyka-za-temoyu-kvalifikaczijnoyi-roboty.pdf": "Практика за темою кваліфікаційної роботи",
    "cs.5729-platforma-ta-programnyj-karkas-hadoop.pdf": "Платформа та програмний каркас Hadoop",
}

OPP_OVERRIDES = {
    "cs.20464-komercializacziya-intelektualnyh-system.pdf": "Системи штучного інтелекту",
    "programuvannya-mikroprocesornyh-prystroyiv-rozumnogo-domu.pdf": "Штучний інтелект",
}

```

```

}

SKIP_MODULES_PARSING = {
    "cs.20464-komerczializacziya-intelektualnyh-system.pdf",
}

def extract_text(pdf_path: Path) -> str:
    text_parts = []
    with pdfplumber.open(pdf_path) as pdf:
        for page in pdf.pages:
            t = page.extract_text(layout=True)
            if t:
                text_parts.append(t)
    return "\n".join(text_parts)

def extract_course_code(filename: str) -> Optional[str]:
    m = re.match(r"(cs\\.d+)", filename)
    return m.group(1) if m else None

def name_from_filename(filename: str) -> str:
    if filename in NAME_OVERRIDES:
        return NAME_OVERRIDES[filename]
    stem = filename.replace(".pdf", "").replace(".txt", "")
    stem = re.sub(r"^cs\\.d+[-_]", "", stem)
    words = stem.replace("_", "-").split("-")
    return " ".join(w.capitalize() for w in words if w).strip()

def extract_field(text: str, field_num: int, next_field_num: int = None) -> str:
    next_num = next_field_num or (field_num + 1)
    start_pattern = rf"^s*{field_num}\\.?s"
    end_pattern = rf"^s*{next_num}\\.?s"
    lines = text.split("\n")
    start_idx = None
    end_idx = len(lines)
    for i, line in enumerate(lines):
        if start_idx is None and re.match(start_pattern, line):
            start_idx = i
        elif start_idx is not None and re.match(end_pattern, line):
            end_idx = i
            break
    if start_idx is None:
        return ""
    return "\n".join(lines[start_idx:end_idx])

def collapse_lines(text: str) -> str:
    return re.sub(r"\s+", " ", text).strip()

def parse_course_name(text: str, filename: str) -> str:
    # Стратегія 0: «Силабус навчальної дисципліни «Назва»»
    m = re.search(
        r"Сил{1,2}абус\s+навчальної\s+дисципліни\s*[\«\»"]{1}([\«\»\''\n]{5,200})[\«\»"]{1}"
        ,
        text[:1500],
        re.IGNORECASE,
    )
    if m:

```

```

return collapse_lines(m.group(1))

field5 = extract_field(text, 5, 6)
if field5:
    body = re.sub(
        r"^s*5\.?s*(?:Код і )?Назва дисципліни\s*(?:\ (інформація з
ЦІСТ\))?\s*",
        "",
        field5,
        flags=re.IGNORECASE,
    )
    body = re.sub(r"Назва дисципліни\s*(?:\ (інформація з ЦІСТ\))?\s*",
        "", body, flags=re.IGNORECASE)
    body = re.sub(r"Код і назва дисципліни\s*(?:\ (інформація з
ЦІСТ\))?\s*", "", body, flags=re.IGNORECASE)
    body = re.sub(r"\(OK[^\)]*\)", "", body)
    body = re.sub(r"\([^\)]*\)", "", body)
    candidate = collapse_lines(body)
    candidate = re.sub(r"^Сил?л?абус[^\n]*", "", candidate).strip()
    if 5 < len(candidate) < 200 and re.match(r"^[А-ЯГІІЄА-Z]",
candidate):
        return candidate

m = re.search(
    r"Назва\s+дисципліни(?:\s*\ (інформація з ЦІСТ\))?\s*\n+\s*([А-
ЯГІІЄА-Z][^\n]+(?:\n\s+[а-ягіїєа-з][^\n]+)*)",
    text[:3000],
)
if m:
    candidate = collapse_lines(re.sub(r"\([^\)]*\)", "", m.group(1)))
    if 5 < len(candidate) < 200:
        # Обрізаю на ключових словах якщо вони з'являються
        for stop in ["Факультет", "Рівень", "Код", "Кількість"]:
            idx = candidate.find(stop)
            if idx > 5:
                candidate = candidate[:idx].strip()
        return candidate

    r"Назва\s+дисципліни\s{2,}([А-ЯГІІЄА-Z][^\n]+)",
    text[:3000],
)
if m:
    candidate = collapse_lines(re.sub(r"\([^\)]*\)", "", m.group(1)))
    if 5 < len(candidate) < 200:
        return candidate

head = text[:600]
for pattern in [
    r"Сил{1,2}абус\s+навчальної\s+дисципліни:\s*",
    r"Назва\s+дисципліни\s*\ (інформація з ЦІСТ\)\s*",
]:
    head = re.sub(pattern, "", head, flags=re.IGNORECASE)
fac_idx = head.find("Факультет")
if fac_idx > 5:
    candidate = collapse_lines(head[:fac_idx])
    if 5 < len(candidate) < 200:
        return candidate

return name_from_filename(filename)

def parse_specialization(text: str, filename: str) -> str:
    if filename in OPP_OVERRIDES:

```

```

        return f"ОПП «{OPP_OVERRIDES[filename]}»"

    field4 = extract_field(text, 4, 5)
    opp = None

    m = re.search(r"ОПП\s*«([^\s]+)»", field4)
    if m:
        opp = m.group(1).strip()

    if not opp:
        m = re.search(r'ОПП\s*["\']([^\s"\']+)["\']', field4)
        if m:
            opp = m.group(1).strip()

    if not opp:
        m = re.search(r"Освітньо-професійна\s+програма\s+([А-ЯҐІІЄА-З][^\n]+)", field4)
        if m:
            opp = m.group(1).strip()

    if not opp:
        m = re.search(r"ОПП\s*«([^\s]+)»", text[:5000])
        if m:
            opp = m.group(1).strip()

    if not opp:
        m = re.search(
            r"Назва\s+освітньої\s+програми\s+([А-ЯҐІІЄА-З][А-ЯҐІІЄа-яґіієА-Зa-z\s]+)?(?:\s{3,}|\n|$)",
            text[:5000]
        )
        if m:
            opp = m.group(1).strip()

    if not opp:
        m = re.search(r"\bОП\s*«([^\s]+)»", text[:5000])
        if m:
            opp = m.group(1).strip()

    if opp:
        opp = re.sub(r"^\s\---+", "", opp).strip()
        opp = re.sub(r"\s+", " ", opp)
        return f"ОПП «{opp}»"

    return ""

def find_prerequisites_section(text: str) -> str:
    m = re.search(
        r"Передумови\s+для\s+навчання[^\n]*\n?\s*(?:за\s+дисципліною)\s*\n?([^\n]+(?:\n\s+[^\n]+){0,5})",
        text,
        re.IGNORECASE,
    )
    if not m:
        return ""

    body = m.group(1)
    for stop in [
        "Анотація", "Компетентності", "Цілі\s+навчання",
        "Структура\s+дисципліни", "Результати\s+навчання",
        "\n\s*\d+\.\s+[А-ЯҐІІЄ]", "Система\s+оцінювання",
    ]:

```

```

        idx = re.search(stop, body)
        if idx:
            body = body[:idx.start()]

    return collapse_lines(body)

def parse_prerequisites(text: str) -> tuple[str, list[str], bool]:
    field9 = extract_field(text, 9, 10)

    if "Передумови" in field9 or "Передумови" in field9.replace(" ", ""):
        body = field9
    else:
        body = find_prerequisites_section(text)
        if not body:
            return "", [], False

    body = re.sub(r"^\s*9\.\?\s*Передумови[^\A-ЯҐІїЄ\n]*\n?", "", body,
        flags=re.MULTILINE)
    body = re.sub(r"^\s*Передумови\s+для\s+навчання[^\n]*\n?", "", body,
        flags=re.MULTILINE)
    body = re.sub(r"за\s+дисципліною\s*", "", body, flags=re.IGNORECASE)
    body = re.sub(
        r"Раніше\s+мають\s+бути\s+вивчен[іо]\s*(?:дисципліни|розділи)?\s*:\s*",
        "",
        body,
        flags=re.IGNORECASE,
    )
    body = collapse_lines(body)

    lower = body.lower()
    school_only = any(
        kw in lower for kw in [
            "знання з інформатики", "навички роботи з комп",
            "у рамках вивчення на рівні середн",
        ]
    )

    if ";" in body:
        parts = [p.strip(" .") for p in body.split(";") if p.strip()]
    else:
        raw_parts = [p.strip(" .") for p in body.split(",") if p.strip()]
        parts = []
        for p in raw_parts:
            if p and parts and re.match(r"^[a-яґіїєа-з]", p):
                parts[-1] = parts[-1] + ", " + p
            else:
                parts.append(p)

    courses = []
    for item in parts:
        item = item.strip(" .")
        if len(item) < 5:
            continue
        item_lower = item.lower()
        if any(kw in item_lower for kw in [
            "середн", "шкіл", "рамках вивчення",
            "знання з інформатики", "навички роботи з комп",
        ]):
            continue
        item = re.sub(r"Раніше\s+мають\s+бути\s+вивчен[іо]\s*", "", item,
            flags=re.IGNORECASE).strip()
        if len(item) > 4:

```

```

        courses.append(item)

    return body, courses, school_only and not courses

def find_annotation_section(text: str) -> str:
    field10 = extract_field(text, 10, 11)
    if field10 and "Анотація" in field10:
        return field10

    m = re.search(
        r"Анотація\s*(?:\s(Зміст\))?\s*\n?\s*(?:дисципліни)\s*\n?([\s\S]+?)(?=\n\s*\nd{1,2}\s*\s*Компетентності|\Z)",
        text,
        re.IGNORECASE,
    )
    if m:
        return m.group(1)

    return ""

def parse_modules_and_topics(text: str) -> tuple[list[Module], str]:
    annotation = find_annotation_section(text)
    if not annotation:
        return [], ""

    annotation = re.sub(
        r"^\s*10\s*\s*Анотація[^\n]*\n",
        "",
        annotation,
        flags=re.MULTILINE,
    )
    annotation = re.sub(r"^\s*\s(Зміст\)\s+дисципліни\s*", "", annotation,
        flags=re.MULTILINE)

    annotation_full = collapse_lines(annotation)

    flat = annotation_full

    markers = []

    for m in re.finditer(
        r"(?:Змістовий\s+)\s*Модуль\s+(\d+)[\.\s:]",
        flat,
        re.IGNORECASE,
    ):
        markers.append((m.start(), "module", int(m.group(1)), m.end()))

    for m in re.finditer(
        r"(?:^|[\.\.;\s])(\d{1,2})\s*\s*([А-ЯҐІІЄА-Z])",
        flat,
    ):
        pos = m.start(1)
        before = flat[max(0, pos - 20):pos]
        if re.search(r"Модуль\s*$", before, re.IGNORECASE):
            continue
        if re.search(r"Тема\s*$", before, re.IGNORECASE):
            continue
        markers.append((m.start(1), "module", int(m.group(1)), m.end(1) +
1))

    for m in re.finditer(

```

```

        r"Тема\s*(\d+)[\.\:\s]+",
        flat,
        re.IGNORECASE,
    ):
        markers.append((m.start(), "topic", int(m.group(1)), m.end()))

    markers.sort(key=lambda x: x[0])

    seen_positions = set()
    unique_markers = []
    for mk in markers:
        if mk[0] in seen_positions:
            continue
        seen_positions.add(mk[0])
        unique_markers.append(mk)
    markers = unique_markers

    modules = []
    current_module = None

    for i, (pos, mtype, num, end) in enumerate(markers):
        next_pos = markers[i + 1][0] if i + 1 < len(markers) else len(flat)
        content = flat[end:next_pos].strip()

        title_match = re.match(r"^([^\.;]{3,200}?)(?:[\.;]|$)", content)
        title = title_match.group(1).strip() if title_match else
content[:100]
        title = title.rstrip(":. ").strip()

        if mtype == "module":
            if not any(mm.number == num and mm.title == title for mm in
modules):
                current_module = Module(number=num, title=title)
                modules.append(current_module)

            elif mtype == "topic":
                if not current_module:
                    current_module = Module(number=1, title="Загальний зміст")
                    modules.append(current_module)
                topic = Topic(number=num, title=title, description=content)
                current_module.topics.append(topic)

    return modules, annotation_full

def parse_competencies(text: str) -> list[Competency]:
    field11 = extract_field(text, 11, 12)
    if not field11 or "Компетентності" not in field11:
        m = re.search(
r"Компетентності[^\n]*\n?([\s\S]+?)(?=\n\s*\d{1,2}\.?\s*Результати|Дата|Сис
тема оцінювання|\Z)",
            text,
            re.IGNORECASE,
        )
        if m:
            field11 = m.group(1)
        else:
            return []

    comps = []
    seen_codes = set()
    matches = re.finditer(

```

```

r"\b(ЗК|ФК|ПР|PH)\s*(\d+)[\.\:]?s+(.+?)(?=\s+(?:ЗК|ФК|ПР|PH)\s*\d|$)",
    field11.replace("\n", " "),
    re.DOTALL,
)
for m in matches:
    prefix = m.group(1)
    num = m.group(2)
    text_c = m.group(3).strip().rstrip(".")
    text_c = re.sub(r"\s+", " ", text_c)
    if len(text_c) > 400:
        text_c = text_c[:400].rsplit(".", 1)[0] + "."
    code = f"{prefix}{num}"
    if code in seen_codes:
        continue
    seen_codes.add(code)
    comps.append(Competency(code=code, text=text_c))

return comps

def parse_syllabus(pdf_path: Path) -> Course:
    text = extract_text(pdf_path)
    code = extract_course_code(pdf_path.name)

    name = parse_course_name(text, pdf_path.name)
    specialization = parse_specialization(text, pdf_path.name)

    level = ""
    field2 = extract_field(text, 2, 3)
    m = re.search(r"Рівень вищої освіти\s*([\n]+)", field2)
    if m:
        level = m.group(1).strip()
    elif "магістерський" in text[:2000].lower() or "магістер" in
text[:2000].lower():
        level = "Другий (магістерський)"
    elif "бакалавр" in text[:2000].lower():
        level = "Перший (бакалаврський)"

    credits = None
    field6 = extract_field(text, 6, 7)
    if not field6:
        m =
re.search(r"Кількість\s+(?:ЄКТС\s+)?кредитів\s*([^\d]*(\d+(?:[\.\,]\d+)?)",
text[:3000])
        if m:
            try:
                credits = float(m.group(1).replace(",", "."))
            except ValueError:
                pass
        else:
            m = re.search(r"(\d+(?:[\.\,]\d+)?)", field6.replace("Кількість
ЄКТС", "").replace("кредитів", ""))
            if m:
                try:
                    credits = float(m.group(1).replace(",", "."))
                except ValueError:
                    pass

    semester = None
    field8 = extract_field(text, 8, 9)
    if not field8:
        m = re.search(r"(\d+[\-\s]*й?\s+(?:рік|курс) [\n]*семестр)",
text[:3000])
        if m:

```

```

        semester = m.group(1).strip()
    else:
        m = re.search(r"(\d+[\-\s]*й?\s+(?:рік|курс) [\^\n]*семестр)",
field8)
        if m:
            semester = m.group(1).strip()

    prereq_raw, prereqs, is_school = parse_prerequisites(text)
    if pdf_path.name in SKIP_MODULES_PARSING:
        modules = []
        annotation = find_annotation_section(text)
        if annotation:
            annotation = re.sub(r"^\s*\d+\.?\s*(?:Цілі|Анотація) [\^\n]*\n",
"", annotation, flags=re.MULTILINE)
            annotation = collapse_lines(annotation)
        else:
            modules, annotation = parse_modules_and_topics(text)
            competencies = parse_competencies(text)

    return Course(
        file=pdf_path.name,
        code=code,
        name=name,
        specialization=specialization,
        level=level,
        credits=credits,
        semester=semester,
        prerequisites_raw=prereq_raw,
        prerequisites=prereqs,
        is_school_only=is_school,
        annotation_full=annotation,
        modules=modules,
        competencies=competencies,
        raw_text="",
    )

def main(
    input_dir: str = "data/syllabi",
    output_file: str = "output/parsed_courses.json",
):
    input_path = Path(input_dir)
    if not input_path.exists():
        print(f"ПОМИЛКА: папка '{input_dir}' не існує!")
        return []

    pdf_files = sorted(input_path.rglob("*.pdf"))
    print(f"Знайдено {len(pdf_files)} файлів")

    courses = []
    errors = []

    for i, pdf in enumerate(pdf_files, 1):
        try:
            course = parse_syllabus(pdf)
            courses.append(course)
            modules_count = len(course.modules)
            topics_count = sum(len(m.topics) for m in course.modules)
            comps_count = len(course.competencies)
            prereq_count = len(course.prerequisites)
            print(
                f"[{i:2d}/{len(pdf_files)}] {course.name[:50]:50s} | "
                f"мод:{modules_count:2d} | тем:{topics_count:3d} | "
                f"комп:{comps_count:2d} | передум:{prereq_count}"
            )

```

```

    )
    except Exception as e:
        errors.append((pdf.name, str(e)))
        print(f"[{i:2d}/{len(pdf_files)}] ПОМИЛКА {pdf.name}: {e}")

output_path = Path(output_file)
output_path.parent.mkdir(parents=True, exist_ok=True)

data = {
    "total_courses": len(courses),
    "errors": errors,
    "courses": [asdict(c) for c in courses],
}
with open(output_path, "w", encoding="utf-8") as f:
    json.dump(data, f, ensure_ascii=False, indent=2)

print(f"\nЗбережено: {output_file}")
print(f"Курсів спасено: {len(courses)}/{len(pdf_files)}")

with_name = sum(1 for c in courses if c.name)
with_opp = sum(1 for c in courses if c.specialization)
with_mods = sum(1 for c in courses if c.modules)
with_topics = sum(1 for c in courses if any(m.topics for m in
c.modules))
with_prereqs = sum(1 for c in courses if c.prerequisites)
with_comps = sum(1 for c in courses if c.competencies)

print(f"\n=== СТАТИСТИКА ПОЛІВ ===")
print(f"З назвою: {with_name}/{len(courses)}")
print(f"З ОПП: {with_opp}/{len(courses)}")
print(f"З модулями: {with_mods}/{len(courses)}")
print(f"З темами: {with_topics}/{len(courses)}")
print(f"З передумовами: {with_prereqs}/{len(courses)}")
print(f"З компетенціями: {with_comps}/{len(courses)}")

if errors:
    print(f"\nПомилки: {len(errors)}")
    for name, err in errors:
        print(f" - {name}: {err}")

return courses

```

ДОДАТОК Б

Код формування онтології

```

import json
import re
from pathlib import Path

from owlready2 import (
    get_ontology, Thing, ObjectProperty, DataProperty,
    FunctionalProperty, InverseFunctionalProperty,
    sync_reasoner_hermit, OwlReadyInconsistentOntologyError,
    locstr,
)
from rapidfuzz import fuzz, process

FUZZY_THRESHOLD = 80

ONTOLOGY_IRI = "http://www.semanticweb.org/volod/ontologies/2026/edu-plan-ontology"

SCHOOL_KEYWORDS = [
    "середн", "шкіл", "рамках вивчення", "знання з інформатики",
    "навички роботи з комп",
]

def normalize_name(name: str) -> str:
    name = name.lower()
    name = re.sub(r"^[^w\s\-'"]", " ", name, flags=re.UNICODE)
    name = re.sub(r"\s+", " ", name).strip()
    return name

PREREQ_CLEANUP_PATTERNS = [
    r"^Раніше\s+(?:мають|повинні|повинна)\s+бути\s+(?:вивчен[io]|вивчена)\s*(?:такі\s+)?дисципліни?\s*:\s*",
    r"^Мають\s+бути\s+вивчен[io]\s+(?:або\s+вивчатися\s+)?дисципліни\s*:\s*",
    r"^такі\s+дисципліни\s*:\s*",
    r"^дисципліни?\s*:\s*",
    r"^вивчатися\s+дисципліни?\s*:\s*",
    r"^\d+\.\s+",
    r"\s+дисципліною\s+",
]

def clean_prereq_text(text: str) -> str:
    cleaned = text
    for pattern in PREREQ_CLEANUP_PATTERNS:
        cleaned = re.sub(pattern, " ", cleaned, flags=re.IGNORECASE)
    cleaned = re.sub(r"\s+", " ", cleaned).strip(".,;:")
    return cleaned

def fuzzy_match_prerequisite(
    prereq_norm: str,
    normalized_names_list: list,
    threshold: int,
) -> tuple:

    r1 = process.extractOne(
        prereq_norm,
        normalized_names_list,
    
```

```

        scorer=fuzz.token_sort_ratio,
    )
    if r1 and r1[1] >= threshold:
        return r1[0], r1[1], "token_sort"

    r2 = process.extractOne(
        prereq_norm,
        normalized_names_list,
        scorer=fuzz.partial_ratio,
    )
    if r2 and r2[1] >= max(threshold + 10, 90):
        return r2[0], r2[1], "partial"

    best = max([r for r in [r1, r2] if r], key=lambda x: x[1],
default=None)
    if best:
        return None, best[1], None
    return None, 0, None

def safe_iri_name(text: str, max_len: int = 50) -> str:
    """
    Конвертую довільний текст в безпечний фрагмент IRI:
    транслітерую кирилицю, замінюю недопустимі символи на _.
    """
    translit = {
        'а': 'a', 'б': 'b', 'в': 'v', 'г': 'h', 'ґ': 'g', 'д': 'd',
        'е': 'e', 'є': 'ie', 'ж': 'zh', 'з': 'z', 'и': 'y', 'і': 'i',
        'ї': 'i', 'й': 'i', 'к': 'k', 'л': 'l', 'м': 'm', 'н': 'n',
        'о': 'o', 'п': 'p', 'р': 'r', 'с': 's', 'т': 't', 'у': 'u',
        'ф': 'f', 'х': 'kh', 'ц': 'ts', 'ч': 'ch', 'ш': 'sh', 'щ': 'shch',
        'ь': '', 'ю': 'iu', 'я': 'ia', "'": '', '': '', 'ё': 'e',
    }
    result = []
    for ch in text.lower():
        if ch in translit:
            result.append(translit[ch])
        elif ch.isalnum() or ch == "_":
            result.append(ch)
        elif ch in "-":
            result.append("_")
    safe = "".join(result)
    safe = re.sub(r"_+", "_", safe).strip("_")
    return safe[:max_len]

def get_safe_code(course: dict) -> str:
    """Повертаю безпечний код курсу - або з поля code, або генеровано з
    імені файлу."""
    code = course.get("code")
    if code:
        return code.replace(".", "")
    fname = course.get("file", "unknown")
    stem = fname.replace(".pdf", "")
    parts = stem.split("-")[:2]
    return "x_" + safe_iri_name("_".join(parts), max_len=20)

def course_iri(course: dict) -> str:
    code = get_safe_code(course)
    name_part = safe_iri_name(course["name"], max_len=40)
    return f"{code}_{name_part}"

```

```

def module_iri(course: dict, module_num: int) -> str:
    code = get_safe_code(course)
    return f"{code}_M{module_num}"

def topic_iri(course: dict, module_num: int, topic_num: int) -> str:
    code = get_safe_code(course)
    return f"{code}_M{module_num}_T{topic_num}"

def specialization_iri(opp_name: str) -> str:
    clean = opp_name.replace("ОПП", "").replace("«", "").replace("»",
    "").strip()
    return f"spec_{safe_iri_name(clean, max_len=40)}"

def is_school_knowledge(text: str) -> bool:
    """Перевірка чи передумова - шкільне знання."""
    t = text.lower()
    return any(kw in t for kw in SCHOOL_KEYWORDS)

def build_ontology(
    json_path: str = "output/parsed_courses.json",
    output_ontology: str = "data/ontology/edu_plan.owx",
    output_report: str = "output/ontology_report.txt",
    fuzzy_threshold: int = FUZZY_THRESHOLD,
):
    with open(json_path, encoding="utf-8") as f:
        data = json.load(f)

    courses_data = data["courses"]
    print(f"Завантажено {len(courses_data)} курсів з JSON")

    onto = get_ontology(ONTOLOGY_IRI)

    report_lines = [
        "=" * 70,
        "ЗВІТ ПОВУДОВИ ОНТОЛОГІЇ ОСВІТИХ ПЛАНІВ",
        "=" * 70,
        "",
        f"Джерело даних: {json_path}",
        f"Кількість курсів: {len(courses_data)}",
        f"Попір fuzzy-зіставлення: {fuzzy_threshold}%",
        "",
    ]

    # Створення схеми
    with onto:
        class Specialization(Thing): pass
        class Course(Thing): pass
        class Module(Thing): pass
        class Topic(Thing): pass
        class Competency(Thing): pass
        class BaseKnowledge(Thing): pass

        class hasPrerequisite(ObjectProperty):
            domain = [Course]
            range = [Course]

        class isPrerequisiteOf(ObjectProperty):
            domain = [Course]
            range = [Course]

```

```

        inverse_property = hasPrerequisite

class containsModule(ObjectProperty):
    domain = [Course]
    range = [Module]

class containsTopic(ObjectProperty):
    domain = [Module]
    range = [Topic]

class developsCompetency(ObjectProperty):
    domain = [Course]
    range = [Competency]

class leadsToSpecialization(ObjectProperty):
    domain = [Course]
    range = [Specialization]

class requiresBaseKnowledge(ObjectProperty):
    domain = [Course]
    range = [BaseKnowledge]

class hasCredits(DataProperty, FunctionalProperty):
    domain = [Course]
    range = [float]

class hasSemester(DataProperty, FunctionalProperty):
    domain = [Course]
    range = [str]

class hasCode(DataProperty, FunctionalProperty):
    domain = [Course]
    range = [str]

class hasLevel(DataProperty, FunctionalProperty):
    domain = [Course]
    range = [str]

class hasFullText(DataProperty, FunctionalProperty):
    domain = [Course]
    range = [str]

class competencyCode(DataProperty, FunctionalProperty):
    domain = [Competency]
    range = [str]

print(" ТБох створено (класи, властивості)")
report_lines.append("СХЕМА (ТБох):")
report_lines.append(f" Класи: Specialization, Course, Module, Topic, Competency, BaseKnowledge")
report_lines.append(f" Object properties: 7")
report_lines.append(f" Data properties: 6")
report_lines.append("")

# Створення індивідів - Specialization
spec_individuals = {}
unique_opps = set()
for c in courses_data:
    if c["specialization"]:
        unique_opps.add(c["specialization"])

for opp_name in unique_opps:
    iri = specialization_iri(opp_name)
    ind = Specialization(iri, namespace=onto)

```

```

ind.label = [locstr(opp_name, lang="uk")]
spec_individuals[opp_name] = ind

print(f"✓ Створено {len(spec_individuals)} спеціалізацій")
report_lines.append(f"СПЕЦІАЛІЗАЦІЇ ({len(spec_individuals)}):")
for opp_name, ind in spec_individuals.items():
    report_lines.append(f"    - {opp_name} → {ind.name}")
report_lines.append("")

# Створення індивідів - Competency
competency_individuals = {}
competency_texts = {}
for c in courses_data:
    for comp in c["competencies"]:
        code = comp["code"]
        if code not in competency_individuals:
            ind = Competency(code, namespace=onto)
            ind.label = [locstr(code, lang="uk")]
            ind.competencyCode = code
            competency_individuals[code] = ind
            competency_texts[code] = comp["text"][:200]

print(f"✓ Створено {len(competency_individuals)} унікальних
компетенцій")
report_lines.append(f"КОМПЕТЕНЦІЇ ({len(competency_individuals)}): {'',
'.join(sorted(competency_individuals.keys()))}")
report_lines.append("")

# Створення індивідів - Course
course_individuals = {}
canonical_names = {}

for c in courses_data:
    iri = course_iri(c)
    ind = Course(iri, namespace=onto)
    ind.label = [locstr(c["name"], lang="uk")]
    if c.get("annotation_full"):
        ind.hasFullText = c["annotation_full"][:2000]
    if c.get("credits"):
        ind.hasCredits = float(c["credits"])
    if c.get("semester"):
        ind.hasSemester = c["semester"]
    if c.get("code"):
        ind.hasCode = c["code"]
    if c.get("level"):
        ind.hasLevel = c["level"]

    if c["specialization"] and c["specialization"] in spec_individuals:
ind.leadstoSpecialization.append(spec_individuals[c["specialization"]])

        for comp in c["competencies"]:
            if comp["code"] in competency_individuals:
ind.developsCompetency.append(competency_individuals[comp["code"]])

        course_individuals[c["file"]] = ind
        canonical_names[c["file"]] = c["name"]

print(f"✓ Створено {len(course_individuals)} курсів (без передумов)")
report_lines.append(f"КУРСИ: {len(course_individuals)}")
report_lines.append("")

```

```

# Створення модулів і тем
total_modules = 0
total_topics = 0
for c in courses_data:
    course_ind = course_individuals[c["file"]]
    for mod in c["modules"]:
        mod_iri = module_iri(c, mod["number"])
        mod_ind = Module(mod_iri, namespace=onto)
        mod_ind.label = [locstr(f"Модуль {mod['number']}"):
{mod['title']}"], lang="uk")]
        course_ind.containsModule.append(mod_ind)
        total_modules += 1

        for topic in mod["topics"]:
            top_iri = topic_iri(c, mod["number"], topic["number"])
            top_ind = Topic(top_iri, namespace=onto)
            top_ind.label = [locstr(f"Тема {topic['number']}"):
{topic['title']}"], lang="uk")]
            mod_ind.containsTopic.append(top_ind)
            total_topics += 1

print(f"✓ Створено {total_modules} модулів і {total_topics} тем")
report_lines.append(f"МОДУЛІ: {total_modules}")
report_lines.append(f"ТЕМИ: {total_topics}")
report_lines.append("")

# Fuzzy-match передумов
name_to_file = {
    normalize_name(name): file_
    for file_, name in canonical_names.items()
}
normalized_names_list = list(name_to_file.keys())

base_knowledge_individuals = {}
match_log = []
unresolved = []
resolved_count = 0
school_count = 0

for c in courses_data:
    course_ind = course_individuals[c["file"]]
    for raw_prereq_text in c["prerequisites"]:
        prereq_text = clean_prereq_text(raw_prereq_text)
        if len(prereq_text) < 4:
            continue
        prereq_norm = normalize_name(prereq_text)

        if is_school_knowledge(prereq_text):
            bk_iri = f"bk_{safe_iri_name(prereq_text, max_len=40)}"
            if prereq_text not in base_knowledge_individuals:
                bk = BaseKnowledge(bk_iri, namespace=onto)
                bk.label = [locstr(prereq_text, lang="uk")]
                base_knowledge_individuals[prereq_text] = bk

course_ind.requiresBaseKnowledge.append(base_knowledge_individuals[prereq_t
ext])

        school_count += 1
        log_code = c.get('code') or c['file'][:15]
        match_log.append(
            f"    [{log_code}] '{prereq_text[:60]}' → SCHOOL
(BaseKnowledge)"
        )
        continue

```

```

        matched_norm, similarity, method = fuzzy_match_prerequisite(
            prereq_norm, normalized_names_list, fuzzy_threshold
        )

        if matched_norm:
            matched_file = name_to_file[matched_norm]

            if matched_file == c["file"]:
                unresolved.append((c, prereq_text, "self-reference"))
                log_code = c.get('code') or c['file'][:15]
                match_log.append(
                    f"    [{log_code}] '{prereq_text[:60]}' → SELF-
REFERENCE (відкинуто)"
                )
                continue

            matched_course = course_individuals[matched_file]
            course_ind.hasPrerequisite.append(matched_course)
            resolved_count += 1
            tag = "★" if similarity >= 90 else "●" if similarity >= 80
        else "o"

        method_tag = f"[{method}]"
        log_code = c.get('code') or c['file'][:15]
        match_log.append(
            f"    [{log_code}] '{prereq_text[:50]}' → "
            f"{canonical_names[matched_file][:40]}"
            f"({similarity:.0f}%) {tag} {method_tag}"
        )
        else:
            bk_iri = f"bk_{safe_iri_name(prereq_text, max_len=40)}"
            if prereq_text not in base_knowledge_individuals:
                bk = BaseKnowledge(bk_iri, namespace=onto)
                bk.label = [locstr(prereq_text, lang="uk")]
                base_knowledge_individuals[prereq_text] = bk

    course_ind.requiresBaseKnowledge.append(base_knowledge_individuals[prereq_t
ext])

    unresolved.append((c, prereq_text, f"no match
(best={similarity:.0f}%)"))
    log_code = c.get('code') or c['file'][:15]
    match_log.append(
        f"    [{log_code}] '{prereq_text[:50]}' → BaseKnowledge "
        f"(best={similarity:.0f}%, нижче порогу)"
    )

print(f"Fuzzy-match виконано: {resolved_count} розв'язано, "
      f"{school_count} шкільних, {len(unresolved)} нерозв'язаних")

report_lines.append("=" * 70)
report_lines.append("FUZZY-МАТЧ ПЕРЕДУМОВ:")
report_lines.append("=" * 70)
report_lines.append(f"Розв'язано (курс → курс): {resolved_count}")
report_lines.append(f"Шкільні знання (BaseKnowledge): {school_count}")
report_lines.append(f"Нерозв'язано (BaseKnowledge):
{len(unresolved)}")
report_lines.append(f"Усього передумов: {resolved_count +
school_count + len(unresolved)}")
report_lines.append("")
report_lines.append("Легенда: ★ = висока впевненість (≥90%), ● =
середня (80-90%), o = низька (<80%)")
report_lines.append("")
report_lines.append("ЖУРНАЛ ЗІСТАВЛЕНЬ:")
report_lines.extend(match_log)

```

```

report_lines.append("")

# Збереження онтології
output_path = Path(output_ontology)
output_path.parent.mkdir(parents=True, exist_ok=True)

onto.save(file=str(output_path), format="rdfxml")
print(f"✓ Онтологію збережено: {output_path}")
report_lines.append("=" * 70)
report_lines.append(f"ОНТОЛОГІЮ ЗБЕРЕЖЕНО: {output_path}")
report_lines.append("=" * 70)

# Запуск Hermit
print("\nЗапуск Hermit reasoner для перевірки узгодженості...")
try:
    with onto:
        sync_reasoner_hermit(infer_property_values=False)
        print("✓ Hermit: онтологія несуперечлива (немає циклів у графі
передумов)")
        report_lines.append("")
        report_lines.append("Hermit REASONER: ✓ Онтологія узгоджена, циклів
не виявлено")
    except OwlReadyInconsistentOntologyError as e:
        print(f"X Hermit: знайдено суперечність! {e}")
        report_lines.append("")
        report_lines.append(f"Hermit REASONER: X ЗНАЙДЕНО СУПЕРЕЧНІСТЬ:
{e}")
    except Exception as e:
        print(f"△ Hermit не вдалося запустити: {e}")
        print(" (Перевірка все одно успішна на рівні структури
онтології)")
        report_lines.append("")
        report_lines.append(f"Hermit REASONER: △ Не вдалося запустити:
{e}")
        report_lines.append(" Можлива причина: Java/JDK не встановлено в
системі.")
        report_lines.append(" Це не блокер – онтологія створена і
відкриється в Protégé,")
        report_lines.append(" де Hermit вже встановлений і може бути
запущений вручну.")

# Збереження звіту
report_path = Path(output_report)
report_path.parent.mkdir(parents=True, exist_ok=True)
with open(report_path, "w", encoding="utf-8") as f:
    f.write("\n".join(report_lines))
print(f"✓ Звіт збережено: {report_path}")

print()
print("=" * 60)
print("ПІДСУМОК")
print("=" * 60)
print(f"Спеціалізацій: {len(spec_individuals)}")
print(f"Курсів: {len(course_individuals)}")
print(f"Модулів: {total_modules}")
print(f"Тем: {total_topics}")
print(f"Компетенцій: {len(competency_individuals)}")
print(f"BaseKnowledge: {len(base_knowledge_individuals)}")
print(f"Передумов (зв.): {resolved_count}")
print(f"Файл онтології: {output_ontology}")
print(f"Звіт: {output_report}")

```

ДОДАТОК В

Код побудови індексу ембедингів

```

import json
from pathlib import Path
from dataclasses import dataclass, field

import numpy as np
from owlready2 import get_ontology
from sentence_transformers import SentenceTransformer

ONTOLOGY_PATH = "data/ontology/edu_plan.owlx"
EMBEDDINGS_OUTPUT = "output/embeddings.npz"
INDEX_METADATA = "output/index_metadata.json"
MODEL_NAME = "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"

@dataclass
class IndexEntry:
    iri: str
    label: str
    entity_type: str
    text_for_embedding: str

def build_course_text(course) -> str:
    parts = []

    if course.label:
        parts.append(str(course.label[0]))

    for spec in course.leadsToSpecialization:
        if spec.label:
            parts.append(str(spec.label[0]))

    if hasattr(course, "hasFullText") and course.hasFullText:
        text = course.hasFullText
        parts.append(text[:1500])

    return " ".join(parts)

def build_topic_text(topic, module=None, course=None) -> str:
    parts = []

    if topic.label:
        parts.append(str(topic.label[0]))
    if module and module.label:
        parts.append(str(module.label[0]))
    if course and course.label:
        parts.append(str(course.label[0]))

    return " ".join(parts)

def build_specialization_text(spec) -> str:
    if spec.label:
        return str(spec.label[0])
    return spec.name

def collect_entries(onto) -> list:

```

```

entries = []

for spec in onto.search(type=onto.Specialization):
    entries.append(IndexEntry(
        iri=spec.name,
        label=str(spec.label[0]) if spec.label else spec.name,
        entity_type="Specialization",
        text_for_embedding=build_specialization_text(spec),
    ))

courses = onto.search(type=onto.Course)
for course in courses:
    entries.append(IndexEntry(
        iri=course.name,
        label=str(course.label[0]) if course.label else course.name,
        entity_type="Course",
        text_for_embedding=build_course_text(course),
    ))

module_to_course = {}
for course in courses:
    for mod in course.containsModule:
        module_to_course[mod.name] = course

for topic in onto.search(type=onto.Topic):
    parent_module = None
    for mod in onto.search(type=onto.Module):
        if topic in mod.containsTopic:
            parent_module = mod
            break

    parent_course = module_to_course.get(parent_module.name) if
parent_module else None

    entries.append(IndexEntry(
        iri=topic.name,
        label=str(topic.label[0]) if topic.label else topic.name,
        entity_type="Topic",
        text_for_embedding=build_topic_text(topic, parent_module,
parent_course),
    ))

return entries

def main():
    print("=" * 60)
    print("ПОБУДОВА ІНДЕКСУ ВЕКТОРИХ ПРЕДСТАВЛЕНЬ")
    print("=" * 60)

    print(f"\nЗавантажую онтологію: {ONTOLOGY_PATH}")
    import owlready2
    ontology_file = Path(ONTOLOGY_PATH).resolve()
    owlready2.onto_path.append(str(ontology_file.parent))
    onto = get_ontology(ontology_file.name).load()
    print(f"✓ Онтологію завантажено")

    print("\nЗбираю сутності для індексації...")
    entries = collect_entries(onto)
    print(f"✓ Зібрано {len(entries)} сутностей:")
    type_counts = {}
    for e in entries:
        type_counts[e.entity_type] = type_counts.get(e.entity_type, 0) + 1

```

```

for t, c in sorted(type_counts.items()):
    print(f"    {t}: {c}")

print(f"\nЗавантажую sentence-transformer модель:")
print(f"    {MODEL_NAME}")
print("    Перший запуск може зайняти 2-5 хвилин (потрібен інтернет)")
model = SentenceTransformer(MODEL_NAME)
print(f"✓ Модель завантажено")

print(f"\nОбчислюю ембединги для {len(entries)} сутностей...")
texts = [e.text_for_embedding for e in entries]
embeddings = model.encode(
    texts,
    show_progress_bar=True,
    convert_to_numpy=True,
    normalize_embeddings=True,
)
print(f"✓ Обчислено ембединги: shape={embeddings.shape},
dtype={embeddings.dtype}")

output_path = Path(EMBEDDINGS_OUTPUT)
output_path.parent.mkdir(parents=True, exist_ok=True)

np.savez_compressed(
    output_path,
    embeddings=embeddings,
    iris=np.array([e.iri for e in entries]),
    labels=np.array([e.label for e in entries]),
    entity_types=np.array([e.entity_type for e in entries]),
)
print(f"\n✓ Індекс збережено: {output_path}")
print(f"    Розмір: {output_path.stat().st_size / 1024:.1f} КБ")

metadata = {
    "model": MODEL_NAME,
    "embedding_dim": int(embeddings.shape[1]),
    "total_entries": len(entries),
    "by_type": type_counts,
}
with open(INDEX_METADATA, "w", encoding="utf-8") as f:
    json.dump(metadata, f, ensure_ascii=False, indent=2)
print(f"✓ Метадані збережено: {INDEX_METADATA}")

print("\n" + "=" * 60)
print("ГОТОВО")
print("=" * 60)
print("\nТепер можна запускати nlp_matcher.py для пошуку:")
print('    python src/nlp_matcher.py "хочу опанувати нейронні мережі"')

if __name__ == "__main__":
    main()

```

ДОДАТОК Г

Код обробки запиту

```

import re
import sys
import json
import argparse
from pathlib import Path
from dataclasses import dataclass, field
from typing import Optional

import numpy as np

ONTOLOGY_PATH = "data/ontology/edu_plan.owx"
EMBEDDINGS_PATH = "output/embeddings.npz"
MODEL_NAME = "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
SPACY_MODEL = "uk_core_news_sm"

CONFIDENCE_THRESHOLD = 0.40

CONFIDENCE_THRESHOLD_NER = 0.25

QUERY_SYNONYMS = {
    # Машинне навчання та ШІ
    "ml": "машинне навчання",
    "ai": "штучний інтелект",
    "dl": "глибинне навчання",
    "nn": "нейронні мережі",
    "rnn": "рекурентні нейронні мережі",
    "cnn": "згорткові нейронні мережі",
    "rl": "навчання з підкріпленням",
    # Обробка природної мови
    "nlp": "обробка природної мови",
    # Комп'ютерний зір
    "cv": "комп'ютерний зір",
    # Аналіз даних
    "ds": "наука про дані",
    "data science": "наука про дані",
    "big data": "великі дані",
    "etl": "інтеграція даних",
    # Розробка
    "oop": "об'єктно-орієнтоване програмування",
    "oop": "об'єктно-орієнтоване програмування",
    "fp": "функціональне програмування",
    "ui": "інтерфейс користувача",
    "ux": "користувацький досвід",
    "api": "програмний інтерфейс",
    "sql": "бази даних",
    "rest": "веб-сервіси",
    # Інше
    "devops": "розробка та експлуатація",
    "git": "контроль версій",
    "os": "операційна система",
    "bd": "база даних",
}

UA_STOPWORDS = {
    "я", "ти", "він", "вона", "ми", "ви", "вони", "мене", "тебе",
    "хотіти", "могти", "робити", "знати", "розуміти", "стати", "бути",
    "опанувати", "вивчити", "вчити", "навчитися", "вивчати", "розібратися",
    "цей", "той", "така", "якийсь", "будь", "трохи", "більш",
    "як", "що", "де", "коли", "чому", "який", "яка", "який",

```

```

"в", "у", "з", "за", "на", "до", "для", "про", "по", "над", "під",
"і", "та", "але", "або", "чи", "не", "ні", "так",
"мені", "себе", "свій", "своя", "усе", "все", "це", "ці",
"потрібно", "треба", "вміти", "розробляти", "робота", "робити",
}

```

```

@dataclass
class ProcessingStep:
    name: str
    description: str
    output: object

```

```

@dataclass
class MatchResult:
    iri: str
    label: str
    entity_type: str
    similarity: float

```

```

@dataclass
class QueryResult:
    original_query: str
    steps: list
    matches: list
    sparql_query: str
    confidence_ok: bool

```

```

class QueryProcessor:

    def __init__(
        self,
        ontology_path: str = ONTOLOGY_PATH,
        embeddings_path: str = EMBEDDINGS_PATH,
        model_name: str = MODEL_NAME,
        spacy_model: str = SPACY_MODEL,
        verbose: bool = False,
    ):
        self.verbose = verbose
        if verbose:
            print("Ініціалізація NLP-процесора...")

            # Лематизатор (spacy для української)
            if verbose:
                print("Завантажую spacy модель для української...")
            import spacy
            try:
                self.nlp = spacy.load(spacy_model)
            except OSError:
                print(f"ПОМИЛКА: модель spacy '{spacy_model}' не знайдена.")
                print(f"Встанови її командою: python -m spacy download {spacy_model}")
                sys.exit(1)

            # Sentence-transformer
            if verbose:
                print("Завантажую sentence-transformer...")
            from sentence_transformers import SentenceTransformer
            self.encoder = SentenceTransformer(model_name)

            # Завантажую заздалегідь обчислений індекс ембедингів

```

```

if verbose:
    print(f" Завантажую індекс ембедингів: {embeddings_path}")
if not Path(embeddings_path).exists():
    print(f"ПОМИЛКА: файл індексу не знайдено: {embeddings_path}")
    print("Запусти спочатку: python src/build_index.py")
    sys.exit(1)

data = np.load(embeddings_path, allow_pickle=True)
self.embeddings = data["embeddings"] # (N, 384)
self.iris = data["iris"]
self.labels = data["labels"]
self.entity_types = data["entity_types"]

# Словник термінів для NER (формуємо з міток онтології)
self._build_term_dictionary()

self.ontology_path = ontology_path

if verbose:
    print(f" ✓ Готово. Індекс містить {len(self.iris)}
сутностей.")

def _build_term_dictionary(self):
    self.domain_terms = set()
    for label in self.labels:
        label_str = str(label).lower()
        for prefix in ["модуль", "тема"]:
            if prefix in label_str:
                parts = label_str.split(":", 1)
                if len(parts) == 2:
                    label_str = parts[1].strip()
                    break
        words = label_str.split()
        for n in (1, 2, 3):
            for i in range(len(words) - n + 1):
                term = " ".join(words[i:i + n])
                if len(term) > 3 and not term.startswith(("опп", "оп
"))):
                    self.domain_terms.add(term)

# ТОКЕНІЗАЦІЯ
def tokenize(self, query: str) -> list:
    doc = self.nlp(query.lower())
    tokens = [tok.text for tok in doc if not tok.is_space]
    return tokens

# ЛЕМАТИЗАЦІЯ
def lemmatize(self, query: str) -> list:
    doc = self.nlp(query.lower())
    lemmas = [
        tok.lemma_ for tok in doc
        if not tok.is_punct and not tok.is_space and len(tok.text) > 1
    ]
    return lemmas

# NER (виявлення іменованих сутностей)
def extract_entities(self, query: str, lemmas: list) -> list:
    entities = []
    seen = set()

    doc = self.nlp(query)
    for ent in doc.ents:
        term = ent.text.strip().lower()
        if term and term not in seen and len(term) > 2:

```

```

        entities.append({"term": ent.text, "type": ent.label_,
"source": "spacy"})
        seen.add(term)

    meaningful_lemmas = [l for l in lemmas if l not in UA_STOPWORDS and
len(l) > 2]
    lemmas_set = set(meaningful_lemmas)

    original_lower = query.lower()
    for term in self.domain_terms:
        if term in original_lower and term not in seen:
            term_words = set(term.split())
            if term_words & lemmas_set or any(w in original_lower for w
in term_words):
                entities.append({"term": term, "type": "DOMAIN",
"source": "dictionary"})
                seen.add(term)

    if not entities and meaningful_lemmas:
        entities = [
            {"term": l, "type": "TOKEN", "source": "fallback"}
            for l in meaningful_lemmas
        ]

    return entities

# ЕМБЕДИНГІ
def encode(self, text: str) -> np.ndarray:
    return self.encoder.encode(
        text,
        convert_to_numpy=True,
        normalize_embeddings=True,
    )

# СЕМАНТИЧНЕ ЗІСТАВЛЕННЯ
def semantic_match(
    self,
    query_vector: np.ndarray,
    top_n: int = 5,
    filter_type: Optional[str] = None,
) -> list:
    scores = self.embeddings @ query_vector # (N,)

    if filter_type:
        mask = self.entity_types == filter_type
        indices = np.where(mask)[0]
        filtered_scores = scores[indices]
        top_local = np.argsort(-filtered_scores)[:top_n]
        top_indices = indices[top_local]
    else:
        top_indices = np.argsort(-scores)[:top_n]

    results = []
    for idx in top_indices:
        results.append(MatchResult(
            iri=str(self.iris[idx]),
            label=str(self.labels[idx]),
            entity_type=str(self.entity_types[idx]),
            similarity=float(scores[idx]),
        ))
    return results

def normalize_query(self, query: str) -> tuple:
    replacements = []

```

```

        normalized = query
        for abbr, full in QUERY_SYNONYMS.items():
            pattern = r"\b" + re.escape(abbr) + r"\b"
            if re.search(pattern, normalized, re.IGNORECASE):
                normalized = re.sub(pattern, full, normalized,
flags=re.IGNORECASE)
                replacements.append((abbr, full))
        return normalized, replacements

def check_ner_match(self, entities: list, course_label: str) -> bool:
    course_lower = course_label.lower()
    for ent in entities:
        term = ent.get("term", "").lower().strip()
        if len(term) < 4:
            continue
        if term in UA_STOPWORDS:
            continue
        if term in course_lower:
            return True
    return False

# Генерація SPARQL
def generate_sparql(self, match: MatchResult) -> str:
    if match.entity_type == "Course":
        sparql = f"""PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
schema#>
PREFIX edu:  <{_namespace()}>

SELECT ?prereqLabel ?moduleLabel ?compCode WHERE {{
    edu:{match.iri} rdfs:label ?courseLabel .
    OPTIONAL {{
        edu:{match.iri} edu:hasPrerequisite ?prereq .
        ?prereq rdfs:label ?prereqLabel .
    }}
    OPTIONAL {{
        edu:{match.iri} edu:containsModule ?module .
        ?module rdfs:label ?moduleLabel .
    }}
    OPTIONAL {{
        edu:{match.iri} edu:developsCompetency ?comp .
        ?comp edu:competencyCode ?compCode .
    }}
}}"""
    elif match.entity_type == "Topic":
        sparql = f"""PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
schema#>
PREFIX edu:  <{_namespace()}>

SELECT ?moduleLabel ?courseLabel WHERE {{
    edu:{match.iri} rdfs:label ?topicLabel .
    ?module edu:containsTopic edu:{match.iri} .
    ?module rdfs:label ?moduleLabel .
    ?course edu:containsModule ?module .
    ?course rdfs:label ?courseLabel .
}}"""
    elif match.entity_type == "Specialization":
        sparql = f"""PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
schema#>
PREFIX edu:  <{_namespace()}>

SELECT ?courseLabel WHERE {{
    ?course edu:leadsToSpecialization edu:{match.iri} .
    ?course rdfs:label ?courseLabel .
}}"""

```

```

else:
    sparql = f"# Тип {match.entity_type} не підтримується"

return sparql.strip()

# ПОВНИЙ PIPELINE
def process(self, query: str, top_n: int = 3) -> QueryResult:
    steps = []
    original_query = query

    normalized_query, replacements = self.normalize_query(query)
    if replacements:
        steps.append(ProcessingStep(
            name="Нормалізація запиту",
            description="Заміна IT-аббревіатур на канонічні форми",
            output=[f"{a} → {f}" for a, f in replacements] +
[f"Результат: «{normalized_query}»"],
        ))
    query = normalized_query

    # Токенізація
    tokens = self.tokenize(query)
    steps.append(ProcessingStep(
        name="Токенізація",
        description="Розбиття запиту на токени",
        output=tokens,
    ))

    # Лематизація
    lemmas = self.lemmatize(query)
    steps.append(ProcessingStep(
        name="Лематизація",
        description="Приведення слів до базової форми",
        output=lemmas,
    ))

    # NER
    entities = self.extract_entities(query, lemmas)
    steps.append(ProcessingStep(
        name="NER (виявлення сутностей)",
        description="Виявлення доменних термінів у запиті",
        output=entities,
    ))

    # Ембединг
    meaningful = [l for l in lemmas if l not in UA_STOPWORDS]
    cleaned_query = " ".join(meaningful) if meaningful else query
    vector = self.encode(cleaned_query)
    steps.append(ProcessingStep(
        name="Ембединг",
        description=f"Векторне представлення розмірністю
{vector.shape[0]}",
        output={"shape": vector.shape, "norm":
float(np.linalg.norm(vector))},
    ))

    # Семантичне зіставлення
    matches = self.semantic_match(vector, top_n=top_n,
filter_type="Course")
    steps.append(ProcessingStep(
        name="Семантичне зіставлення",
        description=f"Топ-{top_n} курсів за косинусною подібністю",
        output=matches,
    ))

```

```

best = matches[0] if matches else None
confidence_ok = False
if best:
    ner_hit = self.check_ner_match(entities, best.label)
    if ner_hit:
        threshold = CONFIDENCE_THRESHOLD_NER
        confidence_ok = best.similarity >= threshold
    else:
        threshold = CONFIDENCE_THRESHOLD
        confidence_ok = best.similarity >= threshold

# SPARQL
sparql = self.generate_sparql(best) if best else "# Курс не
знайдено"
steps.append(ProcessingStep(
    name="Генерація SPARQL",
    description="Формальний запит до онтології",
    output=sparql,
))

return QueryResult(
    original_query=original_query,
    steps=steps,
    matches=matches,
    sparql_query=sparql,
    confidence_ok=confidence_ok,
)

def _namespace() -> str:
    return "http://www.semanticweb.org/volod/ontologies/2026/edu-plan-
ontology#"

def print_result(result: QueryResult):
    print()
    print("=" * 70)
    print(f"ЗАПИТ: «{result.original_query}»")
    print("=" * 70)

    total_steps = len(result.steps)
    for i, step in enumerate(result.steps, 1):
        print(f"\n[{i}/{total_steps}] {step.name}")
        print(f"        {step.description}")

        if isinstance(step.output, list):
            if step.name.startswith("Семантичне"):
                for j, m in enumerate(step.output, 1):
                    tag = "★" if m.similarity >= 0.7 else "●" if
m.similarity >= 0.5 else "○"
                    print(f"        {j}. {m.label[:60]}
({m.similarity:.3f}) {tag}")
            else:
                preview = step.output[:10]
                print(f"        {preview}")
                if len(step.output) > 10:
                    print(f"        ... та ще {len(step.output) - 10}")
        elif isinstance(step.output, dict):
            for k, v in step.output.items():
                print(f"        {k}: {v}")
        elif isinstance(step.output, str):
            for line in step.output.split("\n"):
                print(f"        {line}")

```

```

        else:
            print(f"                {step.output}")

    print()
    print("-" * 70)
    if result.confidence_ok:
        print(f"✓ ЗНАЙДЕНО З ВПЕВНЕНІСТЮ: {result.matches[0].label}")
        print(f"    Подібність: {result.matches[0].similarity:.3f}")
        print(f"    IRI: {result.matches[0].iri}")
    else:
        best_sim = result.matches[0].similarity if result.matches else 0
        print(f"△ ВПЕВНЕНІСТЬ НИЗЬКА (поріг {CONFIDENCE_THRESHOLD})")
        print(f"    Найкращий збір: {result.matches[0].label if result.matches else 'немає'}")
        print(f"    Подібність: {best_sim:.3f}")
        print(f"    Можливо, запит занадто загальний або не стосується предметної галузі.")
    print("-" * 70)

def main():
    parser = argparse.ArgumentParser(
        description="NLP-зіставлення запиту користувача з онтологією освітніх планів"
    )
    parser.add_argument("query", type=str, help="Запит природною мовою")
    parser.add_argument("--top-n", type=int, default=3, help="Скільки кандидатів показувати")
    parser.add_argument("--json", action="store_true", help="Вивід у JSON-форматі")
    args = parser.parse_args()

    qp = QueryProcessor(verbose=not args.json)
    result = qp.process(args.query, top_n=args.top_n)

    if args.json:
        out = {
            "query": result.original_query,
            "best_match": {
                "label": result.matches[0].label if result.matches else None,
                "iri": result.matches[0].iri if result.matches else None,
                "similarity": result.matches[0].similarity if result.matches else 0,
            },
            "all_matches": [
                {"label": m.label, "iri": m.iri, "similarity": m.similarity}
                for m in result.matches
            ],
            "confidence_ok": result.confidence_ok,
            "sparql": result.sparql_query,
        }
        print(json.dumps(out, ensure_ascii=False, indent=2))
    else:
        print_result(result)

if __name__ == "__main__":
    main()

```

ДОДАТОК Д

Код побудови плану

```

import sys
import argparse
from pathlib import Path
from dataclasses import dataclass, field
from typing import Optional
from collections import defaultdict, deque

import owlready2
from owlready2 import get_ontology

ONTOLOGY_PATH = "data/ontology/edu_plan.owx"

@dataclass
class PlanNode:
    iri: str
    label: str
    semester: Optional[str] = None
    credits: Optional[float] = None
    code: Optional[str] = None
    direct_prerequisites: list = field(default_factory=list)
    depth: int = 0
    dependents_count: int = 0
    importance: str = "optional" # "critical" | "important" | "optional"

@dataclass
class BaseKnowledgeNode:
    iri: str
    label: str

@dataclass
class LearningPlan:
    target_iri: str
    target_label: str
    ordered_sequence: list = field(default_factory=list)
    skipped_known: list = field(default_factory=list)
    base_knowledge: list = field(default_factory=list)
    total_credits: float = 0.0
    error_message: Optional[str] = None
    status: str = "normal"
    related_courses: list = field(default_factory=list)

class LearningPlanBuilder:

    def __init__(self, ontology_path: str = ONTOLOGY_PATH, verbose: bool = False):
        self.verbose = verbose
        if verbose:
            print(f"Завантажую онтологію: {ontology_path}")

        ontology_file = Path(ontology_path).resolve()
        owlready2.onto_path.append(str(ontology_file.parent))
        self.onto = get_ontology(ontology_file.name).load()

        if verbose:
            courses = list(self.onto.search(type=self.onto.Course))

```

```

        print(f"✓ Онтологія завантажена. Курсів: {len(courses)}")

    def find_course_by_label(self, label_query: str):
        all_courses = list(self.onto.search(type=self.onto.Course))

        for c in all_courses:
            if c.label and str(c.label[0]).lower() == label_query.lower():
                return c

        query_lower = label_query.lower()
        for c in all_courses:
            if c.label and query_lower in str(c.label[0]).lower():
                return c

        return None

    # DFS - збір транзитивних передумов
    def collect_prerequisites_dfs(self, target_course):
        base_knowledge = set()
        depth_map = {}

        stack = [(target_course, 0)]

        while stack:
            current, depth = stack.pop()

            if self.onto.BaseKnowledge in current.is_a:
                base_knowledge.add(current)
                continue

            if self.onto.Course not in current.is_a:
                continue

            if current in all_courses:
                if depth > depth_map[current]:
                    depth_map[current] = depth
                continue

            all_courses.add(current)
            depth_map[current] = depth

            for prereq in current.hasPrerequisite:
                stack.append((prereq, depth + 1))

            for bk in current.requiresBaseKnowledge:
                base_knowledge.add(bk)

        return all_courses, depth_map, base_knowledge

    # Топологічне сортування (алгоритм Кана)
    def topological_sort_kahn(self, courses):
        course_set = set(courses)
        in_degree = {c: 0 for c in courses}
        adj = defaultdict(list)

        for c in courses:
            for prereq in c.hasPrerequisite:
                if prereq in course_set:
                    adj[prereq].append(c)
                    in_degree[c] += 1

        queue = deque([c for c in courses if in_degree[c] == 0])
        ordered = []

```

```

while queue:
    current = queue.popleft()
    ordered.append(current)

    for next_course in adj[current]:
        in_degree[next_course] -= 1
        if in_degree[next_course] == 0:
            queue.append(next_course)

is_valid = len(ordered) == len(courses)
return ordered, is_valid

# Адаптація під профіль користувача
def adapt_to_profile(self, ordered_courses, known_labels):
    if not known_labels:
        return ordered_courses, []

    known_labels_lower = [k.lower().strip() for k in known_labels]

    final = []
    skipped = []
    for c in ordered_courses:
        course_label = str(c.label[0]).lower() if c.label else
c.name.lower()

        is_known = False
        for known in known_labels_lower:
            if known in course_label or course_label in known:
                is_known = True
                break

        if is_known:
            skipped.append(c)
        else:
            final.append(c)

    return final, skipped

def _build_target_already_known_plan(self, target_course,
known_courses: list) -> LearningPlan:
    target_label = str(target_course.label[0]) if target_course.label
else target_course.name

    related = []
    for dependent in target_course.isPrerequisiteOf:
        if self.onto.Course in dependent.is_a:
            label = str(dependent.label[0]) if dependent.label else
dependent.name
            related.append(PlanNode(
                iri=dependent.name,
                label=label,
                credits=float(dependent.hasCredits) if
dependent.hasCredits else 0.0,
                code=dependent.hasCode,
            ))

    return LearningPlan(
        target_iri=target_course.name,
        target_label=target_label,
        ordered_sequence=[],
        skipped_known=[],
        base_knowledge=[],
        total_credits=0.0,
        is_valid=True,

```

```

        status="target_known",
        related_courses=related,
    )

    def build_plan(
        self,
        target_course,
        known_courses: Optional[list] = None,
    ) -> LearningPlan:
        target_label = str(target_course.label[0]) if target_course.label
    else target_course.name
        known_courses = known_courses or []

        target_label_lower = target_label.lower()
        for known in known_courses:
            known_lower = known.lower().strip()
            if known_lower in target_label_lower or target_label_lower in
known_lower:
                return self._build_target_already_known_plan(target_course,
known_courses)

        if self.verbose:
            print(f"\nЦільовий курс: {target_label}")
            print(f"Запускаю DFS-обхід графа передумов...")

        # DFS
        all_courses, depth_map, base_knowledge =
self.collect_prerequisites_dfs(target_course)

        if self.verbose:
            print(f" Знайдено {len(all_courses)} курсів у графі
передумов")
            print(f" Знайдено {len(base_knowledge)} шкільних передумов")

        # Топологічне сортування
        ordered, is_valid = self.topological_sort_kahn(list(all_courses))

        if not is_valid:
            return LearningPlan(
                target_iri=target_course.name,
                target_label=target_label,
                is_valid=False,
                error_message="Виявлено цикл у графі передумов - план
неможливо побудувати.",
            )

        if self.verbose:
            print(f" Топологічне сортування дало {len(ordered)} курсів")

        # Адаптація під профіль
        final_sequence, skipped = self.adapt_to_profile(ordered,
known_courses or [])

        if self.verbose and skipped:
            print(f" Виключено {len(skipped)} вже відомих курсів")

        final_set = set(final_sequence)
        dependents_count = {c: 0 for c in final_sequence}
        for c in final_sequence:
            for prereq in c.hasPrerequisite:
                if prereq in final_set:
                    dependents_count[prereq] += 1

        plan_nodes = []

```

```

total_credits = 0.0
for c in final_sequence:
    direct_prereqs = [
        p.name for p in c.hasPrerequisite
        if p in final_set
    ]
    credits = float(c.hasCredits) if c.hasCredits else 0.0
    total_credits += credits

    dep_count = dependents_count[c]
    if dep_count >= 3:
        importance = "critical"
    elif dep_count >= 1:
        importance = "important"
    else:
        importance = "optional"

    plan_nodes.append(PlanNode(
        iri=c.name,
        label=str(c.label[0]) if c.label else c.name,
        semester=c.hasSemester,
        credits=credits,
        code=c.hasCode,
        direct_prerequisites=direct_prereqs,
        depth=depth_map.get(c, 0),
        dependents_count=dep_count,
        importance=importance,
    ))

skipped_nodes = [
    PlanNode(
        iri=c.name,
        label=str(c.label[0]) if c.label else c.name,
        semester=c.hasSemester,
        credits=float(c.hasCredits) if c.hasCredits else 0.0,
        code=c.hasCode,
    )
    for c in skipped
]

bk_nodes = [
    BaseKnowledgeNode(
        iri=bk.name,
        label=str(bk.label[0]) if bk.label else bk.name,
    )
    for bk in base_knowledge
]

status = "normal"
if len(plan_nodes) == 1 and len(skipped_nodes) >= 1:
    if plan_nodes[0].iri == target_course.name:
        status = "all_known"

return LearningPlan(
    target_iri=target_course.name,
    target_label=target_label,
    ordered_sequence=plan_nodes,
    skipped_known=skipped_nodes,
    base_knowledge=bk_nodes,
    total_credits=total_credits,
    is_valid=True,
    status=status,
)

```

```

def print_plan(plan: LearningPlan):
    print()
    print("=" * 70)
    print(f"ОСВІТНІЙ ПЛАН: «{plan.target_label}»")
    print("=" * 70)

    if not plan.is_valid:
        print(f"\nX ПОМИЛКА: {plan.error_message}")
        return

    if plan.base_knowledge:
        print(f"\n└ БАЗОВІ ЗНАННЯ (мають бути на початку):")
        for bk in plan.base_knowledge:
            print(f"|      • {bk.label}")
        print(f"└")

    if plan.skipped_known:
        print(f"\n[ВЖЕ ЗАСВОЄНО - пропускаємо ({len(plan.skipped_known)} курсів):]")
        for node in plan.skipped_known:
            print(f"      ✓ {node.label}")

    n_critical = sum(1 for n in plan.ordered_sequence if n.importance == "critical")
    n_important = sum(1 for n in plan.ordered_sequence if n.importance == "important")
    n_optional = sum(1 for n in plan.ordered_sequence if n.importance == "optional")

    print(f"\n└ ПОСЛІДОВНІСТЬ ВИВЧЕННЯ ({len(plan.ordered_sequence)} курсів):")
    for i, node in enumerate(plan.ordered_sequence, 1):
        is_target = i == len(plan.ordered_sequence)

        if is_target:
            icon = "🎯"
            label_prefix = "ЦІЛЬОВИЙ КУРС: "
        else:
            if node.importance == "critical":
                icon = "🔴"
            elif node.importance == "important":
                icon = "🟡"
            else:
                icon = "🟢"
            label_prefix = ""

        print(f"|   {i:2d}. {icon} {label_prefix}{node.label}")

        if is_target:
            pass
        else:
            n = node.dependents_count
            if n == 1:
                form = "1 інший курс плану"
            elif 2 <= n <= 4:
                form = f"{n} інші курси плану"
            else:
                form = f"{n} інших курсів плану"

            if n >= 3:
                print(f"|               критичний - від нього залежать {form}")
            elif n >= 1:

```

```

        if n == 1:
            print(f"|          важливий - від нього залежить {form}")
        else:
            print(f"|          важливий - від нього залежать {form}")
    else:
        print(f"|          опційний - не є передумовою для інших
курсів плану")
        print(f"└")

    print(f"\nЗАГАЛОМ:")
    print(f"  Курсів у плані: {len(plan.ordered_sequence)}")
    print(f"    ● критичних:  {n_critical}")
    print(f"    ● важливих:   {n_important}")
    print(f"    ● опційних:   {n_optional}")
    if plan.skipped_known:
        print(f"  Пропущено (відомі): {len(plan.skipped_known)} курсів")
    print()
    print("Порада: починайте з ● критичних - без них подальші курси будуть
незрозумілими.")
    print("    ● опційні можна вивчати в довільному порядку.")
    print()

def main():
    parser = argparse.ArgumentParser(
        description="Побудова персонального освітнього плану через обхід
графа передумов"
    )
    parser.add_argument("target", type=str, help="Назва цільового курсу")
    parser.add_argument(
        "--known",
        nargs="*",
        default=[],
        help="Список курсів, які користувач уже вивчив",
    )
    args = parser.parse_args()

    builder = LearningPlanBuilder(verbose=True)

    target = builder.find_course_by_label(args.target)
    if not target:
        print(f"X Не знайдено курс з назвою: «{args.target}»")
        print(f"  Спробуй ввести точнішу назву.")
        sys.exit(1)

    plan = builder.build_plan(target, known_courses=args.known)

    print_plan(plan)

if __name__ == "__main__":
    main()

```

ДОДАТОК Е

Код формування пояснень

```

import sys
import argparse
from pathlib import Path
from dataclasses import dataclass, field
from typing import Optional

import owlready2
from owlready2 import get_ontology

from graph_traversal import LearningPlanBuilder, LearningPlan, PlanNode

ONTOLOGY_PATH = "data/ontology/edu_plan.owlx"

COMPETENCY_DESCRIPTIONS = {
    "ЗК1": "абстрактне мислення, аналіз та синтез",
    "ЗК2": "застосування знань у практичних ситуаціях",
    "ЗК3": "знання та розуміння предметної галузі",
    "ЗК4": "комунікація державною мовою",
    "ЗК5": "комунікація іноземною мовою",
    "ЗК6": "здатність до навчання та оволодіння сучасними знаннями",
    "ЗК7": "пошук, обробка та аналіз інформації",
    "ЗК8": "критичне та самокритичне мислення",
    "ЗК9": "робота в команді",
    "ФК1": "теоретичні засади комп'ютерних наук",
    "ФК2": "виявлення статистичних закономірностей, обчислювальний інтелект",
    "ФК3": "логічне мислення, формальні моделі, проектування алгоритмів",
    "ФК4": "формалізація предметної галузі",
    "ФК6": "робота з даними (включно з великими)",
    "ФК7": "архітектурні рішення комп'ютерних систем",
    "ФК8": "проектування програмного забезпечення",
    "ФК10": "командна робота над проектом",
    "ФК11": "інтелектуальний аналіз даних",
    "ФК12": "якість IT-проектів",
    "ФК13": "процеси розробки програмного забезпечення",
    "ФК14": "експлуатація та реінжиніринг ПЗ",
    "ФК15": "сучасні тенденції розвитку комп'ютерних наук",
    "ФК16": "інноваційний характер IT-проектів",
    "ФК17": "аналіз бібліографічних джерел",
}

}

@dataclass
class CourseExplanation:
    iri: str
    label: str
    importance: str
    reason: str = ""
    enables: list = field(default_factory=list)
    requires: list = field(default_factory=list)
    competencies: list = field(default_factory=list)
    path_to_target: list = field(default_factory=list)

class Explainer:

    def __init__(self, ontology_path: str = ONTOLOGY_PATH, verbose: bool = False):
        self.verbose = verbose

```

```

if verbose:
    print(f"Завантажую онтологію: {ontology_path}")

ontology_file = Path(ontology_path).resolve()
owlready2.onto_path.append(str(ontology_file.parent))
self.onto = get_ontology(ontology_file.name).load()

if verbose:
    print("✓ Онтологія завантажена")

def _get_competency_label(self, code: str) -> str:
    return COMPETENCY_DESCRIPTIONS.get(code, "")

def _find_individual_by_iri(self, iri: str):
    return self.onto.search_one(iri=f"*{iri}")

def _build_path_to_target(self, course_iri: str, plan_iris: list) ->
list:
    target_iri = plan_iris[-1] if plan_iris else None
    if not target_iri or course_iri == target_iri:
        return []

    from collections import deque
    visited = {course_iri}
    queue = deque([(course_iri, [course_iri])])
    plan_set = set(plan_iris)

    while queue:
        current_iri, path = queue.popleft()
        current = self._find_individual_by_iri(current_iri)
        if not current:
            continue

        for dependent in current.isPrerequisiteOf:
            if dependent.name not in plan_set:
                continue
            if dependent.name in visited:
                continue
            new_path = path + [dependent.name]
            if dependent.name == target_iri:
                labels = []
                for iri in new_path:
                    ind = self._find_individual_by_iri(iri)
                    if ind and ind.label:
                        labels.append(str(ind.label[0]))
                return labels
            visited.add(dependent.name)
            queue.append((dependent.name, new_path))

    return []

def explain_course(
    self,
    node: PlanNode,
    plan: LearningPlan,
    is_target: bool = False,
) -> CourseExplanation:
    course = self._find_individual_by_iri(node.iri)
    if not course:
        return CourseExplanation(
            iri=node.iri,
            label=node.label,
            importance="unknown",
            reason="Курс не знайдено в онтології",

```

```

    )

    plan_iris = [n.iri for n in plan.ordered_sequence]
    plan_set = set(plan_iris)

    enables = []
    for dependent in course.isPrerequisiteOf:
        if dependent.name in plan_set:
            if dependent.label:
                enables.append(str(dependent.label[0]))
    requires = []
    for prereq in course.hasPrerequisite:
        if prereq.name in plan_set:
            if prereq.label:
                requires.append(str(prereq.label[0]))
    competencies = []
    for comp in course.developsCompetency:
        code = str(comp.label[0]) if comp.label else comp.name
        description = self._get_competency_label(code)
        competencies.append((code, description))
    if is_target:
        importance = "target"
        n_prereqs = len(requires)
        reason = (
            f"Це ваш цільовий курс - кінцева мета навчального плану. "
            f"Для його успішного засвоєння необхідно попередньо вивчити
"
            f"{n_prereqs} {self._declension(n_prereqs, 'курс', 'курси',
'курсів')}}."
        )
    elif node.importance == "critical":
        importance = "critical"
        n = node.dependents_count
        reason = (
            f"Критичний базовий курс - на ньому ґрунтуються "
            f"{n} {self._declension(n, 'наступний курс', 'наступні
курси', 'наступних курсів')}} "
            f"вашого плану. Без нього подальше навчання буде
неможливим."
        )
    elif node.importance == "important":
        importance = "important"
        n = node.dependents_count
        reason = (
            f"Важлива передумова для "
            f"{n} {self._declension(n, 'наступного курсу', 'наступних
курсів', 'наступних курсів')}} "
            f"вашого плану."
        )
    else:
        importance = "optional"
        reason = (
            "Опційний компонент плану - він не є передумовою для інших
курсів, "
            "але розширює компетенції в межах вашої цілі."
        )

    path_to_target = []
    if not is_target:
        path_to_target = self._build_path_to_target(node.iri,
plan_iris)

    return CourseExplanation(
        iri=node.iri,

```

```

        label=node.label,
        importance=importance,
        reason=reason,
        enables=enables,
        requires=requires,
        competencies=competencies,
        path_to_target=path_to_target,
    )

    @staticmethod
    def _declension(n: int, form1: str, form24: str, form5: str) -> str:
        last_two = n % 100
        if 11 <= last_two <= 14:
            return form5
        last = n % 10
        if last == 1:
            return form1
        if 2 <= last <= 4:
            return form24
        return form5

    def explain(self, plan: LearningPlan) -> list:
        if not plan.is_valid:
            return []

        explanations = []
        last_index = len(plan.ordered_sequence) - 1
        for i, node in enumerate(plan.ordered_sequence):
            is_target = i == last_index
            ex = self.explain_course(node, plan, is_target=is_target)
            explanations.append(ex)
        return explanations

    def print_explanations(plan: LearningPlan, explanations: list):
        print()
        print("=" * 70)
        print(f"ПОЯСНЕННЯ ОСВІТНЬОГО ПЛАНУ: «{plan.target_label}»")
        print("=" * 70)

        if plan.base_knowledge:
            print(f"\n📖 БАЗОВІ ЗНАННЯ (припускається, що користувач їх має):")
            for bk in plan.base_knowledge:
                print(f"    • {bk.label}")
            print(f"    Ці передумови виходять за межі ОПП і вивчаються до вступу.")

        for ex in explanations:
            print()
            if ex.importance == "target":
                icon = "🎯"
            elif ex.importance == "critical":
                icon = "🔴"
            elif ex.importance == "important":
                icon = "🟡"
            else:
                icon = "🟢"
            print(f"📚 {icon} {ex.label}")
            print(f"    " + "-" * 65)

            print(f"    Чому в плані: {ex.reason}")

            if ex.requires:
                print(f"\n    Спирається на курси з плану:")

```

```

        for r in ex.requires:
            print(f"        ← {r}")

    if ex.enables and ex.importance != "target":
        print(f"\n    Уможливорює подальші курси:")
        for e in ex.enables:
            print(f"        → {e}")

    if ex.competencies:
        print(f"\n    Розвиває компетенції:")
        with_desc = [(c, d) for c, d in ex.competencies if d]
        without_desc = [c for c, d in ex.competencies if not d]

        shown = with_desc[:5]
        for code, desc in shown:
            print(f"        • {code}: {desc}")
        remaining_with = [c[0] for c in with_desc[5:]]
        all_remaining = remaining_with + without_desc
        if all_remaining:
            print(f"        • також: {' '.join(all_remaining)}")

    if ex.path_to_target and 2 <= len(ex.path_to_target) <= 5:
        print(f"\n    Шлях до цілі через граф онтології:")
        chain = " → ".join(ex.path_to_target)
        print(f"        {chain}")

print()
print("=" * 70)
print(f"Усі пояснення сформовані з формальних зв'язків онтології.")
print(f"Це гарантує верифікованість – кожне твердження підкріплене")
print(f"конкретним фактом у графі знань.")
print("=" * 70)
print()

def main():
    parser = argparse.ArgumentParser(
        description="Формування пояснень для освітнього плану (XAI)"
    )
    parser.add_argument("target", type=str, help="Назва цільового курсу")
    parser.add_argument(
        "--known",
        nargs="*",
        default=[],
        help="Список курсів, які користувач уже вивчив",
    )
    args = parser.parse_args()

    builder = LearningPlanBuilder(verbose=False)
    target = builder.find_course_by_label(args.target)
    if not target:
        print(f"X Не знайдено курс з назвою: «{args.target}»")
        sys.exit(1)

    plan = builder.build_plan(target, known_courses=args.known)
    if not plan.is_valid:
        print(f"X Помилка побудови плану: {plan.error_message}")
        sys.exit(1)

    explainer = Explainer(verbose=False)
    explanations = explainer.explain(plan)

    print_explanations(plan, explanations)

```

ДОДАТОК Ж

Відомість кваліфікаційної роботи

[illegible]