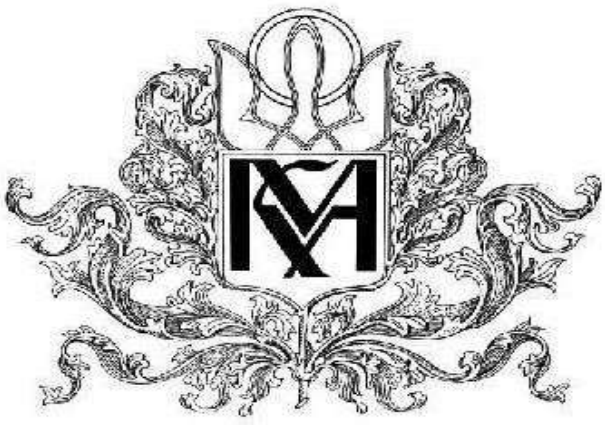
Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики



Аналіз та розв’язання геометричної задачі в онтологічній базі знань

Текстова частина до кваліфікаційної роботи
за спеціальністю „Комп’ютерні науки ” 122

Керівник курсової роботи
Жежерун Олександр Петрович
_____
(підпис)
“____” _____ 2024 р.

Виконала студентка
Прудніков А.О.
“____” _____ 2024 р.

Київ 2024

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав. Кафедри
мультимедійних
систем, доцент, к.ф.-м.н.
О.П.Жежерун

_____
(підпис)
“____”_____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

студентки Прудніковій Анастасії Олександрівни факультету інформатики 4-го курсу

ТЕМА Аналіз та розв’язання геометричної задачі в онтологічній базі знань

Зміст ТЧ до кваліфікаційної роботи:
Індивідуальне завдання
Зміст
Вступ
Розділ 1: Опис онтологічної бази знань
Розділ 2: Теоретичні аспекти геометричних задач у онтологічних базах знань
Розділ 3: Реалізація та розв’язання обраних задач
Розділ 4: Розробка експертної системи
Висновки
Список використаних джерел

Дата видачі “___” _____ 2024 р. Керівник _____
(підпис)

Завдання отримав _____
(підпис)

ЗМІСТ

Анотація

Дана дипломна робота присвячена дослідженню та розробці онтологічної бази знань для вирішення геометричних задач. У розділі 1 надається огляд онтологічної бази знань, включаючи її визначення, структуру, та методи представлення знань. Розділ 2 зосереджується на теоретичних аспектах геометричних задач у онтологічних базах знань, а саме на основних підходах до аналізу та розв'язання таких задач, моделюванні геометричних об'єктів, та використанні допоміжних інструментів. У розділі 3 описана реалізація та розв'язання обраних геометричних задач, представлені результати вирішення трьох конкретних завдань. Розділ 4 спрямований на інтегрування виведених знань у експертну систему. Ця робота має на меті розширення розуміння процесу вирішення геометричних задач за допомогою онтологічних баз знань та впровадження цих підходів у практичні застосування.

Вступ

В сучасному світі, де обсяг інформації зростає експоненційно, важливо розробляти ефективні та інтелектуальні системи для обробки та аналізу цих даних. Одним зі способів вирішення цієї проблеми є застосування онтологічних баз знань, які дозволяють представляти інформацію в структурованому вигляді та робити висновки на основі цієї інформації. Об'єктом дослідження даної роботи є онтологічна база знань для вирішення геометричних задач. Вибір цієї теми обумовлений актуальністю проблеми автоматизації процесу розв'язання геометричних завдань та необхідністю розробки інтелектуальних систем, що здатні до аналізу та розв'язання таких задач, які допоможуть учням в школі швидше опанувати матеріал та спростити процес розв’язування задач.

Метою даного дослідження є розробка онтологічної бази знань та вивчення можливостей її застосування для розв'язання геометричних завдань. Конкретні завдання дослідження включають аналіз теоретичних аспектів геометричних задач, моделювання геометричних об'єктів у онтологічних базах знань та реалізацію алгоритмів для вирішення практичних задач з використанням цієї бази.

Практичне значення дослідження полягає у можливості створення інтелектуальної системи з використанням розробленої онтології, яка здатна до автоматизованого аналізу та розв'язання геометричних задач.

Розділ 1. Опис онтологічної бази знань

1.1 Визначення онтологічної бази знань

Онтологія є ключовим поняттям у представленні знань, особливо в штучному інтелекті та інформатиці. Це стосується формалізації знань та їх обробки в машинах, забезпечуючи структурований і стандартизований спосіб визначення та представлення відносин і зв'язків між різними об'єктами в певній предметній області. Онтології використовуються для представлення знань про предметну область інтересів, забезпечуючи загальну концептуалізацію предметної області, яка є формальною, однозначною та легко обробляється машиною.

Онтології можна використовувати в різних прикладних контекстах, таких як розуміння природної мови, інформаційні системи та системи, засновані на знаннях, і можуть допомогти обмінюватися знаннями про стратегії міркування або методи вирішення проблем. У контексті семантичної мережі онтології є частиною стеку стандартів W3C(World Wide Web Consortium), які забезпечують взаємодію баз даних, пошук між базами даних і безперебійне управління знаннями.

Основні принципи і цілі створення онтологічної бази включають в себе кілька ключових аспектів, які є основоположними для розвитку і управління знаннями. Онтологічна інженерія відіграє вирішальну роль у цьому процесі, впливаючи на те, як знання структуруються та використовуються в різних сферах.

Основні принципи створення онтологічної бази:

1. Реалізм: точне описання реальності для узгодження з модельованою областю.
2. Перспективалізм: визнання різних способів представлення однієї реальності.
3. Фалібілізм: прийняття постійного перегляду та вдосконалення онтологій.
4. Адекватизм: визнання складності реальності та потреби в інклюзивних уявленнях.

Онтологічна інженерія в комп’ютерній науці та інформатиці є новою галуззю, яка вивчає методи та методики побудови онтологій: формальні представлення набору понять і зв’язки між цими поняттями. Створюючи онтології, які відповідають встановленим принципам і стандартам, онтологічна інженерія сприяє сумісності між різноманітними системами, сприяючи безперешкодному обміну знаннями та інтеграції. Редактори онтологій — це програми, призначені для допомоги у створенні або маніпулюванні онтологіями, тоді як навчання онтологій — це автоматичне або напівавтоматичне створення онтологій, включаючи вилучення термінів домену з тексту природної мови.

Найпопулярніші веб-редактори онтологій :

1. Protégé - відомий редактор онтологій, який має веб-версію. Він надає зручний інтерфейс для створення та редагування онтологій, підтримує різні мови онтологій (наприклад, OWL, RDF), а також пропонує функції спільної роботи над онтологіями з team.Website: Protégé Web

2. WebVOWL - це веб-інструмент візуалізації онтологій. Хоча це не повноцінний редактор, як Protégé, він чудово підходить для візуалізації та дослідження існуючих онтологій. Він надає інтерактивне графічне представлення ontologies.Website: WebVOWL

3. TopBraid Composer Maestro Edition: Maestro Edition пропонує безкоштовне веб-середовище для редагування онтологій з деякими обмеженнями. Воно відоме своєю підтримкою SHACL і SPARQL, що робить його придатним для більш просунутого моделювання онтологій.Website: TopBraid Composer Maestro Edition.

У даній роботі більш детально буде розглянуто роботу з Protégé.

1.2 Експертні системи

Експертні системи - це комп'ютерні програми, що імітують здатність людей-експертів приймати рішення в певних галузях. Такі системи використовують методи штучного інтелекту для вирішення складних проблем шляхом міркувань через сукупність знань, представлених переважно у вигляді правил типу «якщо-то».

Експертні системи складаються з двох основних підсистем:

- механізму виведення
- бази знань

Механізм виведення - це важливий компонент, який застосовує логічні правила до бази знань, щоб виводити нову інформацію або приймати рішення. Це частина експертної системи, що приймає рішення, здатна інтерпретувати та оцінювати факти, для того щоб робити логічні висновки. Механізм виведення працює, зіставляючи правила з відомими фактами, вибираючи, які правила застосувати, і виконуючи ці правила, щоб генерувати нові знання або висновки. Цей ітеративний процес триває доти, доки не буде отримано жодної нової інформації.

База знань зберігає факти і правила, в той час як механізм виведення застосовує ці правила до відомих фактів для виведення нової інформації. Експертні системи призначені для того, щоб доповнювати експертів-людей, а не замінювати їх, і будуються за допомогою процесу, який називається інженерією знань, де експерти надають інформацію для бази знань. Ці системи мають такі компоненти, як база знань, механізм виведення та інтерфейс користувача, і для ефективного функціонування вони покладаються на методології представлення знань.

1.3 Структура та основні компоненти онтологічної бази

Основні складові онтологічної бази:

- **Individuals** - це екземпляри або об'єкти, які є основними, "базовими" компонентами онтології. Вони можуть включати конкретні об'єкти, такі як люди, тварини, столи, автомобілі, молекули і планети, а також абстрактні особи, такі як числа і слова.
- **Classes -** абстрактні групи, колекції або набори об'єктів. Класи використовуються для категоризації осіб та інших класів на основі їхніх спільних характеристик.
- **Relations** - це способи, за допомогою яких класи та сутності можуть бути пов'язані між собою. Зв'язки можуть містити атрибути, які уточнюють зв'язок.
- **Function terms** - це складні структури, утворені з певних відношень, які можна використовувати замість окремих термінів у висловлюванні. Вони можуть допомогти виразити більш складні зв'язки між особами та класами.
- **Restrictions** - це формально сформульовані описи того, що має бути істинним для того, щоб деяке твердження було прийняте як вхідні дані. Вони можуть бути використані для визначення обмежень на відносини між особами та класами.
- **Rules** - це твердження у формі речення "якщо-то" (антецедент-консеквент), які описують логічні висновки, що можуть бути зроблені на основі зв'язків між індивідами та класами. Вони можуть бути використані для вираження більш складних відносин між особами та класами.
- **Axioms** - це твердження (включаючи правила) в логічній формі, які разом складають загальну теорію та логіку, яку описує онтологія. Вони можуть бути використані для вираження більш складних зв'язків між особами та класами.
- **Events** - це зміни атрибутів або відношень. Їх можна використовувати для вираження змін у часі у відносинах між особами та класами.

- **Actions** - це типи подій. Їх можна використовувати для вираження більш складних зв'язків між особами та класами.

Ці компоненти, як правило, організовані в ієрархічну структуру, з класами та особами, пов'язаними відносинами підпорядкування, а також атрибутами, відношеннями та функціями, що використовуються для опису характеристик та зв'язків між ними. Обмеження, правила та аксіоми можуть бути використані для визначення обмежень на зв'язки між особами та класами, а події та дії можуть бути використані для вираження змін у цих зв'язках з плином часу

Розробка онтології зазвичай включає процес визначення цих компонентів та їх взаємозв'язків на основі спільного розуміння предметної області, що моделюється, а також використання інструментів та методологій розробки онтологій для забезпечення того, щоб онтологія була добре структурованою, узгодженою та придатною для використання.

1.4 Методи представлення знань в онтологічній базі

Є багато різних способів, якими можна представляти знання. Найпоширеніші з них базуються на семантичних мережах, правилах і логіці, відповідно до сучасних технологій і варіантів використання семантичн. Представлення у вигляді графів RDF і карток знань, що містять структури семантичних мереж, але певна бізнес-логіка зазвичай формалізується у вигляді правил з інтерпретацією "якщо-то", таких як бізнес-правила або формалізми логічного програмування. Точна семантична інтерпретація для кожного з інших типів реалізується за допомогою логіки. Формалізми на основі логіки забезпечують формальну семантику для мов представлення знань, що відкриває шлях до автоматизованої дедукції. Приклади таких мов є OWL та RDF.

OWL (Web Ontology Language) - це семантична веб-мова, призначена для представлення багатих і складних знань про речі, групи речей і зв'язки між ними. Ця мова, заснована на обчислювальній логіці, що дозволяє використовувати знання, виражені нею, в комп'ютерних програмах, наприклад, для перевірки узгодженості або для того, щоб зробити неявні знання явними.

Документи OWL, відомі як онтології, можуть бути опубліковані у всесвітній павутині і можуть посилатися на інші онтології OWL або посилатися на них. OWL надає стандартизований спосіб опису понять, відношень та інших компонентів онтології, що дозволяє формально представляти знання у машинозчитуваному форматі.

OWL має високу виразність, що дозволяє представляти складні та тонкі ідеї про предметну область, виходячи за межі можливостей традиційних мов моделювання, таких як XSD, UML та SQL.

RDF(Resource Description Framework) - це загальна основа для представлення взаємопов'язаних даних в Інтернеті. Вона забезпечує послідовний спосіб опису та обміну метаданими про веб-ресурси, що дозволяє RDF представляти зв'язки між ресурсами у структурованому, машинозчитуваному вигляді. Також це фундаментальною технологією для семантичного вебу, що дозволяє обмінюватися даними і повторно використовувати їх в межах додатків, підприємств і спільнот.

RDF розширює структуру зв'язків в Інтернеті, використовуючи URI для позначення зв'язків між об'єктами, а також двох кінців зв'язку. Це формує спрямований, маркований граф, який можна використовувати для представлення та запиту даних.

Формати серіалізації RDF: XML, Turtle, JSON-LD і N-Triples.

Розділ 2. Теоретичні аспекти геометричних задач у онтологічних базах знань

2.1 Основні підходи до аналізу та розв’язку геометричних задач

Існує два методи для пошуку розв’язку задачі: аналітичний та синтетичний. Аналітичним називається метод, що полягає у міркуваннях від невідомих, шуканих до даних величин. Синтетичний – це метод міркувань від даних до шуканих, невідомих величин. Синтетичний метод варто використовувати якщо задача легка або вже відомий спосіб її розв’язування. Спосіб розв’язування більшості задач, зокрема складніших, треба шукати аналітичним методом, виходячи з вимоги задачі. Для того щоб розв’язати важку задачу, то використовують аналітико-синтетичний метод: прокладають шлях і від вимоги, і від умови, поки не буде знайдено зв’язку між ними.

У даній частині розглядаються основні види задач з геометрії та їх характеристики.

Розглянемо три найпоширеніші види:

- задачі на обчислення;
- задачі на доведення;
- задачі на побудову.

Кожен з цих видів задач вимагає від студента різних навичок та підходів до розв'язання.

Задачі на обчислення в геометрії передбачають розрахунок певної геометричної величини, такої як площа, об'єм, довжина, кут або відстань між точками. Ці задачі часто вимагають від студента використання формул та математичних властивостей геометричних об'єктів для отримання точних числових результатів.

Задачі на доведення засновані на тому, щоб показати, що певна геометрична співвідносність або властивість справджується за допомогою розумових або логічних маніпуляцій. Вони вимагають від студента виявлення логічних зв'язків та використання відомих геометричних тверджень для обґрунтування своїх висновків.

У задачах на побудову студенту потрібно намалювати геометричну фігуру згідно з вказаними умовами. Це може включати побудову прямокутника з відомими сторонами, побудову перпендикуляра або бісектриси до відрізка, або побудову трикутника за трьома сторонами або кутами. Також відносимо сюди задачі, що розв’язуються за допомогою додаткової побудови. У таких задачах студент повинен виявити креативність та вміння застосовувати геометричні конструкції для досягнення вказаної мети.

Отже у даній роботі буде досліджено та проаналізовано два види задач: задачі на обчислення та задачі на доведення.

2.2 Моделювання геометричних об'єктів у онтологічних базах знань

Моделювання геометричних об'єктів в онтологічних базах знань передбачає представлення геометричних понять, зв'язків і властивостей у структурований і формальний спосіб, зрозумілий комп'ютерам. Онтології забезпечують основу для організації знань шляхом визначення класів, властивостей і зв'язків між об'єктами.

По-перше, спочатку необхідно визначити класи для базових геометричних об'єктів. Таких як точки, лінії, кола, багатокутники. Кожен клас повинен мати властивості, які описують його характеристики.

Створення класу Трикутників у Protégé виглядатиме наступним чином:

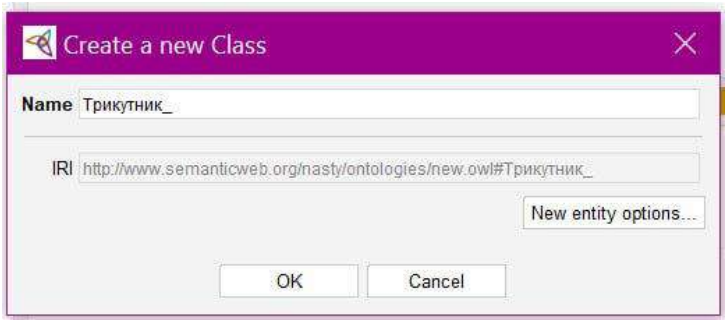


Рисунок 2.1 Створення нового класу

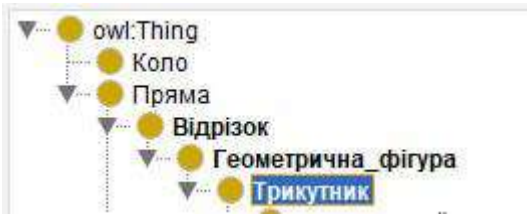


Рисунок 2.2 Створений трикутник

Будь-які маніпуляції та дії, що пов’язанні з класами у Protégé відбуваються у Entities Tab.

Проте для конкретного виду трикутника, наприклад Прямокутний трикутник, потрібно вказати який саме трикутник вважатиметься Прямокутним. Тому необхідно додати визначення:

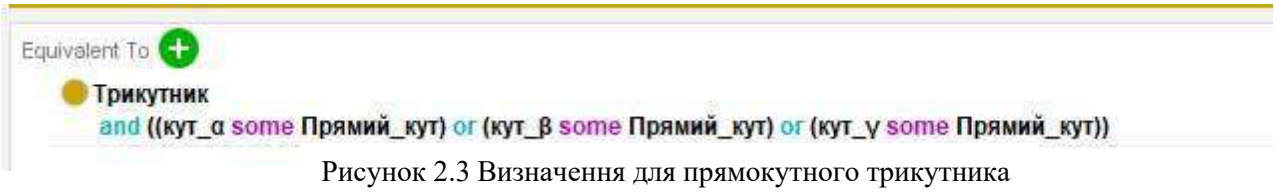


Рисунок 2.3 Визначення для прямокутного трикутника

Отже, за даним визначенням Трикутник буде вважатись прямокутним, коли один з його кутів буде прямим.

Також можна експортувати дану онтологію у форматі OWL. Наступне зображення ілюструє створений в попередньому кроці клас у коді.

Рисунок 2.4 Клас трикутника у RDF

Код у форматі OWL чи RDF автоматично розділений на підрозділи, що робить його читабельним та інтуїтивно зрозумілим.

Рисунок 2.5 Анотації до створених елементів у

Важливо визначити властивості, що описуватимуть геометричні об'єкти. Ці властивості можуть включати градусні міри кутів, довжини, площі, об'єми та будь-які інші відповідні атрибути. Наприклад, коло може мати такі властивості, як радіус, діаметр, довжина та площа.

Наприклад трикутник матиме наступні властивості:

Рисунок 2.6 Створені властивості

Для сторін та кутів будуть визначені окремі властивості:

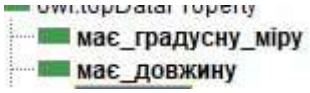


Рисунок 2.7 Створені властивості

У Protégé є окремий Tab для роботи з числовими властивостями – Data Property.

Наступний крок передбачає створення зв'язків між геометричними об'єктами. Наприклад, відрізок може бути частиною багатокутника або коло може перетинатися з лінією. Ці зв'язки можна представити за допомогою таких властивостей, як "частина", "перетинається", "торкається" тощо.

Для створення зв’язків необхідно перейти до Object Property Tab.
Щоб сказати, що трикутник має кути потрібно створити окремі зв’язки, оскільки кути є окремою сутністю.

Рисунок 2.8 Числові властивості

Наприклад: запис для трикутника DEF з двома відомими кутами D та E виглядатиме наступним чином:

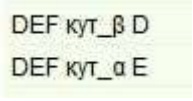


Рисунок 2.9 Трикутник

2.3 Допоміжні інструменти для розв’язання геометричних задач у онтологічних базах знань

Редактор SWRL - це розширення до Protégé-OWL, яке дозволяє інтерактивне редагування правил SWRL. Редактор можна використовувати для створення нових правил SWRL, редагування існуючих правил, а також для читання і запису правил SWRL. Редактор SWRL доступний як вкладка у програмі Protégé-OWL.

SWRL - це специфікація мови з чітко визначеною семантикою, але для виконання правил SWRL розробники повинні реалізувати механізм правил або зіставити його з існуючим.

Синтаксис SWRL дозволяє виражати правила в термінах концепцій OWL, таких як класи, властивості та сутності. Наприклад, правило "якщо людина має рідного брата, який є чоловіком, то ця людина має брата" може бути виражене так:

$$Person(?p) \wedge hasSibling(?p,?s) \wedge Man(?s) \rightarrow hasBrother(?p,?s)$$

Однак, з правилами SWRL у Protégé можуть виникнути деякі проблеми. Міст SWRL не враховує всі обмеження OWL, тому можливі суперечності між правилами та онтологією. Підтримка узгодженості вимагає від користувача періодичного запуску логічного аналізатора. Інтегрований логічний аналізатор і рушій правил був би ідеальним рішенням, але поточний підхід з використанням Pellet підтримує лише основні вбудовані бібліотеки SWRL.

У Protege Reasoner(логічний вивід) - це компонент, який використовується для виведення нових знань на основі аксіом і тверджень, наявних в онтології. Reasoner взаємодіє з онтологією для обчислення висновків, таких як визначення типів для індивідів або обчислення тверджень про властивості. Reasoner Protege можна налаштувати для відображення різних типів висновків, що дозволяє

користувачам налаштовувати інформацію, яка буде відображатись у користувацькому інтерфейсі. Protege підтримує різні різонери, кожен з яких має свої можливості та особливості.

Protege підтримує як локальні, так і розподілені аналізатори. Локальні логічні аналізатори працюють безпосередньо у середовищі Protege, забезпечуючи можливості виведення у реальному часі. З іншого боку, розподілені логічні аналізатори можуть бути використані для великих онтологій або складних завдань логічного виведення і можуть працювати поза середовищем Protege.

Protege підтримує декілька різонерів, наприклад найпопулярнішими є Pellet, HermiT, Fact++.

- Pellet

- Pellet може бути ефективним для обробки великих онтологій з великою кількістю аксіом та класів. Він має ефективні алгоритми для класифікації та перевірки консистентності великих моделей.
- Pellet підтримує OWL 2, останню версію мови онтологій веб-семантики, тому він може бути корисним для роботи з онтологіями, які використовують сучасні функції OWL 2.

- HermiT

- HermiT відомий своєю високою продуктивністю та швидкодією. Він може бути відмінним вибором для великих та складних онтологій, де швидкість роботи має значення.

- Fact++

- Fact++ відомий своєю низьким використанням пам'яті та швидкістю.
- Fact++ може бути корисним для широкого спектру застосувань, від простих до складних онтологій, оскільки він є загальним та ефективним.

При перевірці та перед застосуванням різонера варто переконатись, що результатом виводу будуть всі важливі компоненти, оскільки відразу після підключення виводитись будуть лише деякі властивості. Для того щоб дозволити різонеру показувати в логічних наслідках потрібні дані, наприклад Data Properties потрібно відкрити Configure reasoner та обрати необхідні елементи:

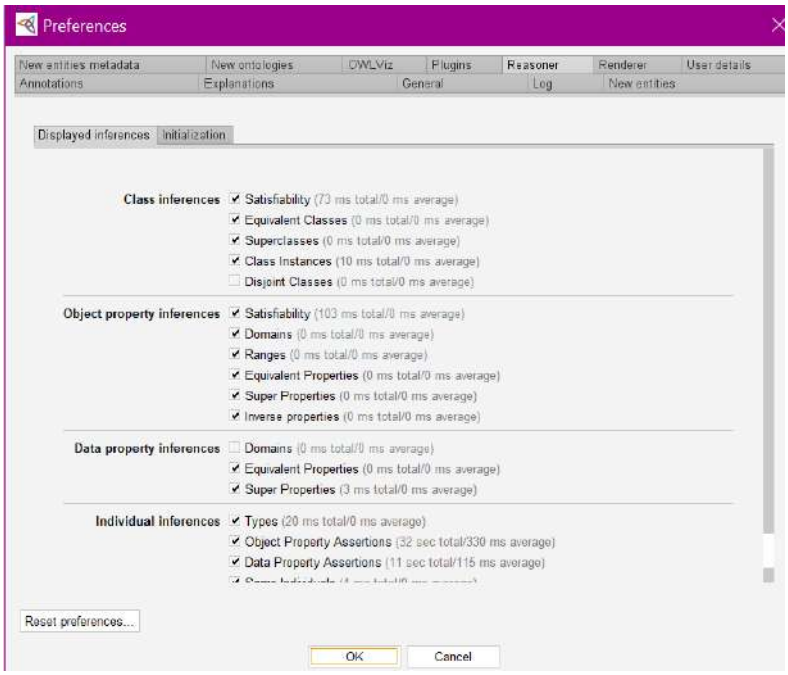


Рисунок 2.10 Підключення виведених різонером властивостей

Логічні міркування у Protege можуть виконувати різні типи виводу, включаючи:

1. Класифікація. Організація індивідів у класи на основі їх властивостей та зв'язків.
2. Перевірка узгодженості. Перевірка того, що онтологія не містить логічних протиріч.
3. Виведення властивостей. Виведення неявних взаємозв'язків між об'єктами на основі аксіом.
4. Пошук екземплярів. Пошук екземплярів, які відповідають певним критеріям, визначеним в онтології.

Різонери у Protege інтегровані у користувацький інтерфейс, що дозволяє користувачам викликати завдання логічного аналізу безпосередньо з інтерфейсу

Protege. Користувачі можуть запускати і зупиняти сеанси логічного висновку, налаштовувати параметри логічного висновку і аналізувати отримані результати в Protege.

Архітектура плагінів Protege дозволяє користувачам розширювати можливості міркувань, інтегруючи додаткові міркування або налаштовуючи існуючі. Ця гнучкість дозволяє користувачам адаптувати функціонал міркувань до своїх конкретних вимог.

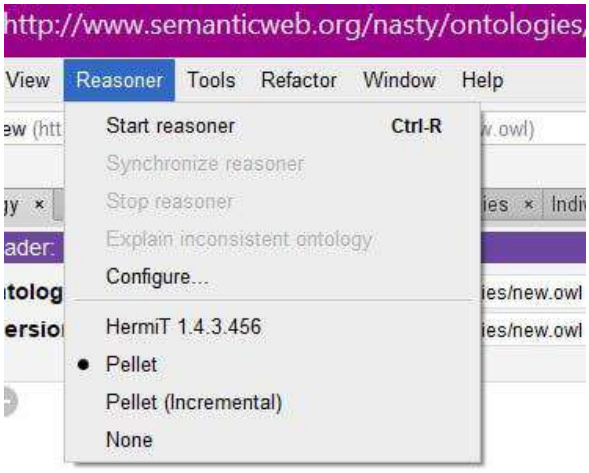


Рисунок 2.11 Різонер

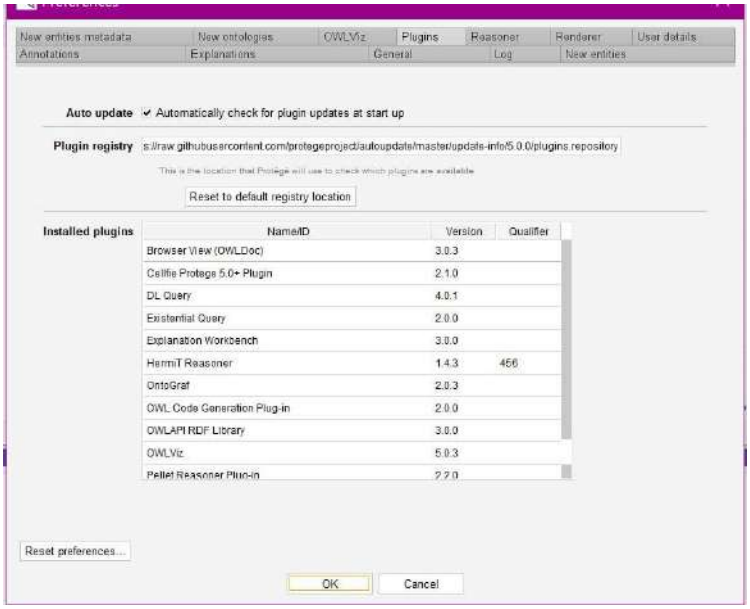


Рисунок 2.12 Плагіни

Різонери відіграють важливу роль у розробці та підтримці онтології в Protege, дозволяючи користувачам використовувати автоматизований висновок для отримання нових знань і забезпечення узгодженості та цілісності онтологічних моделей. Редактор SWRL у Protégé надає можливість створювати та редагувати правила SWRL. Однак, управління узгодженістю між правилами та онтологією вимагає від користувача ретельної уваги.

У Protege, якщо натиснути на значок "?" біля нового виведеного знання (інференції),

Рисунок 2.13 Пояснення до висновків

буде надано детальні відомості про те, як саме це знання було виведене або інферовано. Такі пояснення зазвичай включають:

- Відображення логічних кроків, які були використані для виведення нового знання. Це може включати правила, підстави або умови, які були використані для логічного виводу.
- Вказівка на джерело даних або фактів, які були використані для інференції. Наприклад, які аксіоми або класи були використані.
- Відображення інших знань або фактів, які можуть бути пов'язані з новим знанням або які використовувалися під час інференції.

Ці детальні відомості допомагають краще зрозуміти та проаналізувати результати інференції, а також виявляти потенційні проблеми або неточності в онтології.

Для розв’язання математичних задач використовується різонер Pellet, оскільки для обрахунків необхідні вбудовані математичні функції і цей різонер єдиний, хто їх підтримує. У даній роботі будуть використовуватись два модулі вбудованих функцій: Built-Ins for Comparisons та Math Built-Ins. Проте асортимент функцій в математичному модулі замалий, оскільки немає важливих функцій, таких як квадратний корінь. Також для знаходження тригонометричних значень кута аргументом треба записувати кут у радіанах, що є незручним, бо кути в задачах задаються переважно в градусах. Тож виникає потреба в окремих обрахункав у правилах, для того щоб перевести градуси в радіани.

Розділ 3. Реалізація та розв’язання обраних задач

3.1 Перша задача на обчислення

Умова: Знайти радіус описаного і вписаного кола в прямокутному трикутнику зі сторонами 15, 20, 25.

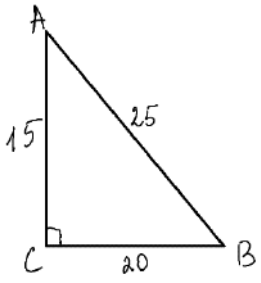


Рисунок 3.1 Задача №1

Для знаходження радіус потрібно дві формули:

$R = \frac{abc}{4S}$ та $r = \frac{S}{p}$, де p – це півпериметр.

Алгоритм розв’язання:

1) Знайти площу та периметр.
2) Підставити знайдені значення та знайти радіуси.

Оскільки трикутник є прямокутним, то щоб знайти площу потрібно знайти половину добутку катетів. Тож створюємо відповідні SWRL правила для площі, периметру та знаходження радіусів:

- Rule: Прямокутний_трикутник(?t), сторона_a(?t, ?s), сторона_b(?t, ?w), сторона_c(?t, ?l), має_довжину(?w, ?l2), має_довжину(?s, ?l1), має_довжину(?l, ?l3), greaterThan(?l3, ?l1), greaterThan(?l3, ?l2), multiply(?res, ?l1, ?l2), divide(?res1, ?res, 2) -> гіпотенуза(?t, ?l), катет(?t, ?s), катет(?t, ?w), площа(?t, ?res1) – Знаходження площі в прямокутному трикутнику.

- Rule: Трикутник(?t), сторона_a(?t, ?s), сторона_b(?t, ?x), сторона_c(?t, ?g), має_довжину(?s, ?l1), має_довжину(?x, ?l2), має_довжину(?g, ?l3), add(?res, ?l1, ?l2, ?l3) -> периметр(?t, ?res) – Знаходження периметру.

- Rule: Трикутник(?t), периметр(?t, ?P), площа(?t, ?s), divide(?p, ?P, 2), divide(?res, ?s, ?p), радіус_вписаного_кола(?t, ?r) -> має_довжину(?r, ?res) – Знаходження довжини радіуса вписаного в коло.

- Rule: Трикутник(?t), сторона_a(?t, ?k), сторона_b(?t, ?l), сторона_c(?t, ?c), має_довжину(?l, ?l2), має_довжину(?c, ?l3), має_довжину(?k, ?l1), площа(?t, ?s), multiply(?dob, ?l1, ?l2, ?l3), multiply(?d, ?s, 4), divide(?res, ?dob, ?d), радіус_описаного_кола(?t, ?r) -> має_довжину(?r, ?res) – Знаходження радіуса описаного кола.

Після запуску різонера з’являються логічні висновки. Вони виділяються жовтим кольором. У трикутнику можна побачити результати обрахування площі та периметра.

Рисунок 3.2 Створений прямокутний трикутник

Довжини радіусів відображаються у властивостях індивідів, що є радіусами в трикутника ABC.

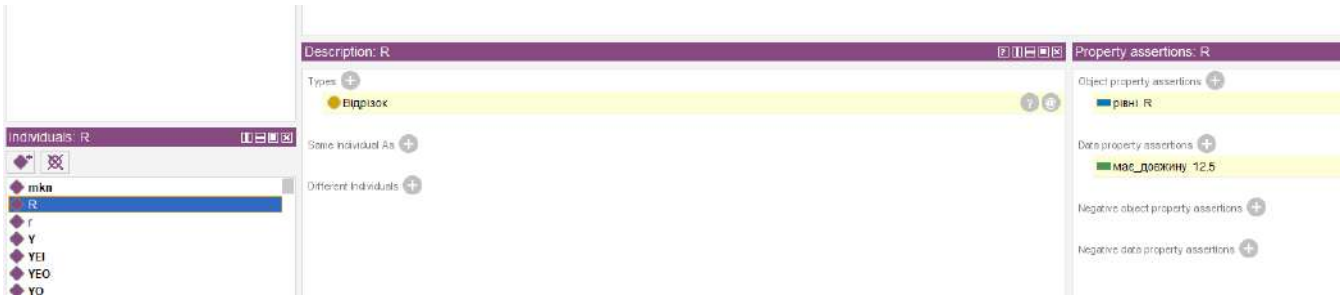


Рисунок 3.3 Знайдений радіус описаного кола

Рисунок 3.4 Знайдений радіус вписаного кола

Отже відповідь R = 12.5, r = 5.

Функція "Export inferred axioms as ontology" у Protege дозволяє експортувати виведенні аксіоми як окрему онтологію. Це корисна функція, яка дозволяє отримати результати логічного виводу у вигляді самостійної онтології, яка містить тільки факти, які були виведені за допомогою різонера.

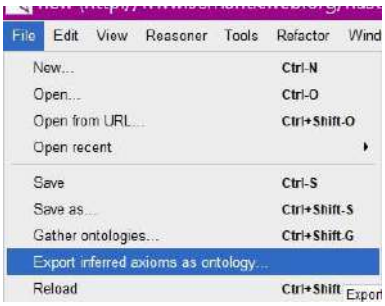


Рисунок 3.5 Експорт виведених висновків

Результати обрахунків радіуса описаного кола:

```
<!-- http://www.semanticweb.org/nasty/ontologies/new.owl#R -->

<owl:NamedIndividual rdf:about="http://www.semanticweb.org/nasty/ontologies/new.owl#R">
    <rdf:type rdf:resource="http://www.semanticweb.org/nasty/ontologies/new.owl#Відрізок"/>
    <rdf:type rdf:resource="http://www.semanticweb.org/nasty/ontologies/new.owl#Пряма"/>
    <new:рівні rdf:resource="http://www.semanticweb.org/nasty/ontologies/new.owl#R"/>
    <new:має_довжину rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">12.5</new:має_довжину>
</owl:NamedIndividual>
```

Рисунок 3.6 Відображення результату в RDF

Результати обрахунків радіуса вписаного кола:

```
<!-- http://www.semanticweb.org/nasty/ontologies/new.owl#r -->

<owl:NamedIndividual rdf:about="http://www.semanticweb.org/nasty/ontologies/new.owl#r">
    <rdf:type rdf:resource="http://www.semanticweb.org/nasty/ontologies/new.owl#Відрізок"/>
    <rdf:type rdf:resource="http://www.semanticweb.org/nasty/ontologies/new.owl#Пряма"/>
    <new:рівні rdf:resource="http://www.semanticweb.org/nasty/ontologies/new.owl#r"/>
    <new:має_довжину rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">5</new:має_довжину>
</owl:NamedIndividual>
```

Рисунок 3.7 Відображення результату в RDF

Розглянемо пояснення та логіку, щодо знаходження великого радіуса:

Рисунок 3.8 Пояснення до розв’язку першої задачі

Отже в цілому алгоритм розв’язання задачі за допомогою онтології збігається з алгоритмом створеним на початку даного підрозділу.

3.2 Друга задача на обчислення

Умова: Висота ВД трикутника АВС поділяє сторону АС на відрізки АД та ДС, АВ = 12 см, $\angle A = 60$, $\angle СВД = 45$. Знайти сторону ДС.

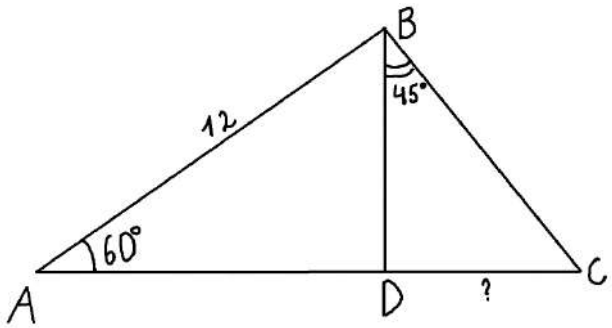


Рисунок 3.9 Задача №2

Алгоритм розв’язання:

1) Створити правило, що висота поділяє трикутник на два прямокутні трикутники.
2) Знайти всі кути в обидвох трикутниках.
3) Для першого трикутника створити правило про кут 30 градусів навпроти якого лежить катет, що дорівнює половині гіпотенузи.

4) Через синус кута 60 градусів знайти сторону ВД.
5) Трикутник ВСД, рівнобедрений за кутами, отже ВД = ДС.

SWRL правила, що необхідні для розв’язання даної задачі:

- Rule: Трикутник(?t), кут_α(?t, ?x), кут_β(?t, ?z), кут_γ(?t, ?g), має_градусну_міру(?x, ?v1), має_градусну_міру(?g, ?v3), add(?res, ?v1, ?v3), subtract(?v2, 180, ?res) -> має_градусну_міру(?z, ?v2) – Знаходження третього кута в трикутнику коли відомі два інші.

- Rule: Трикутник(?t), Трикутник(?r), Трикутник(?q), має_висоту(?t, ?в), сторона_b(?r, ?в), сторона_b(?q, ?в), кут_γ(?r, ?k), кут_γ(?q, ?k) -> має_градусну_міру(?k, 90) – Утворення двох прямокутних трикутників в трикутнику в якому проведено висоту.

- Rule: Трикутник(?т), сторона_a(?т, ?a), сторона_c(?т, ?c), має_довжину(?c, ?д), кут_α(?т, ?к), має_градусну_міру(?к, 30), кут_γ(?т, ?к1), має_градусну_міру(?к1, 90), divide(?рез, ?д, 2) -> має_довжину(?a, ?рез) – Знаходження довжини катету, що лежить навпроти кута в 30 градусів у прямокутному трикутнику.

- Rule: Трикутник(?t), кут_γ(?t, ?к1), має_градусну_міру(?к1, 90), сторона_b(?t, ?v), сторона_c(?t, ?d), має_довжину(?d, ?l3), кут_β(?t, ?y), має_градусну_міру(?y, ?q), divide(?g, 3.14, 180), multiply(?r, ?q, ?g), sin(?s, ?r), multiply(?res, ?s, ?l3) -> має_довжину(?v, ?res) – знаходження сторони в прямокутному трикутнику через синус кута.

- Rule: Трикутник(?t), кут_α(?t, ?a), кут_β(?t, ?b), рівні(?a, ?b) -> Рівнобедрений_трикутники(?t) – Визначення рівнобедреного трикутника за двома рівними кутами.

- Rule: Рівнобедрений_трикутники(?t), сторона_a(?t, ?a), сторона_b(?t, ?b) -> рівні(?a, ?b) – Наслідок, що в рівнобедреного трикутника є дві рівні сторони.

- Rule: Відрізок(?g), Відрізок(?k), має_довжину(?g, ?d1), має_довжину(?k, ?d2), subtract(0, ?d1, ?d2) -> рівні(?g, ?k) – Визначення рівних відрізків.

Відповідь:

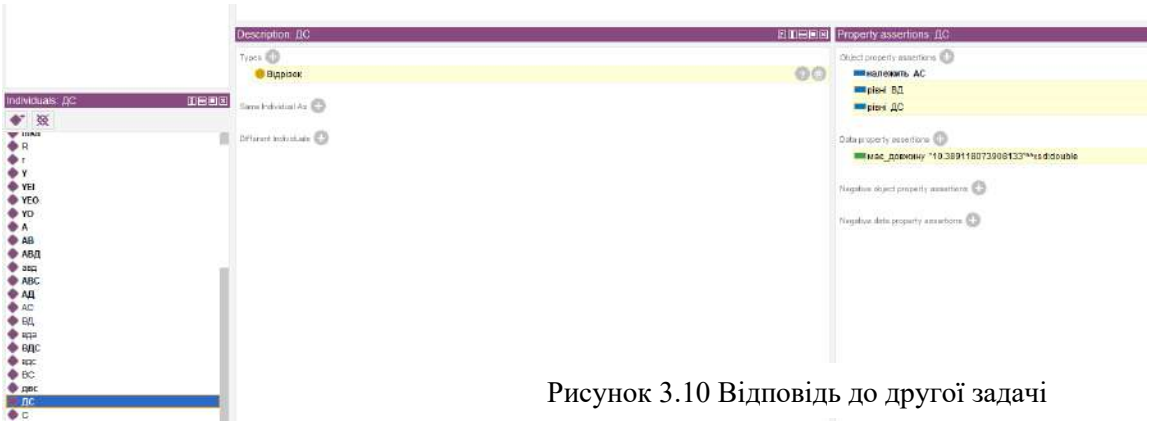


Рисунок 3.10 Відповідь до другої задачі

Отже відповідь 10.39 см.

Задача розв’язується в 5 кроків, тому після запуску різонера необхідно зачекати деякий час, перед тим як з’являться висновки. Так само треба чекати, щоб отримати пояснення до розв’язку.

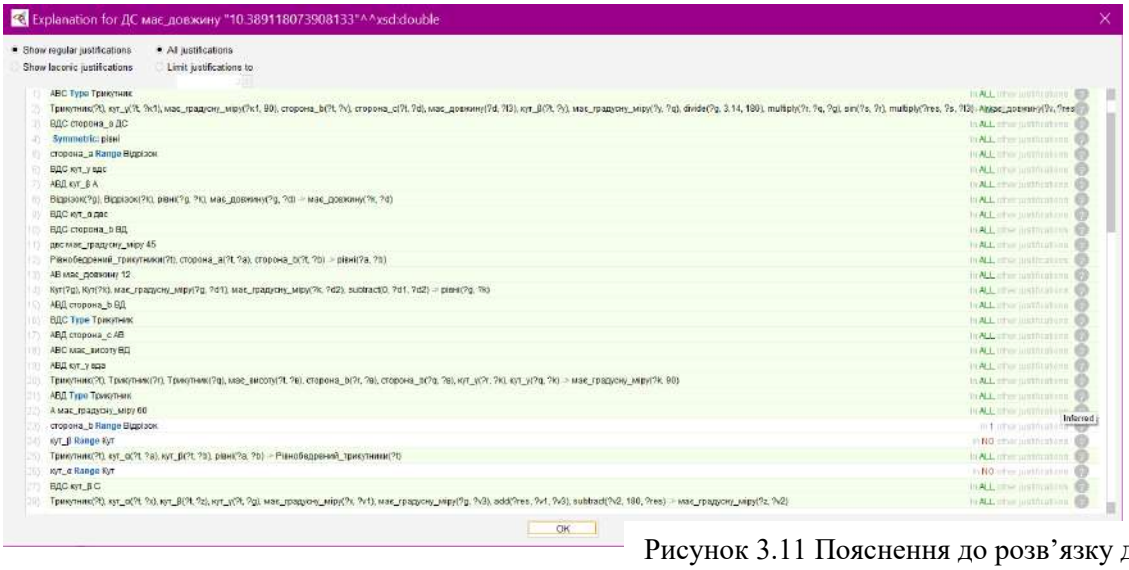


Рисунок 3.11 Пояснення до розв’язку другої задачі

3.3 Третя задача на доведення

Умова: Довести рівність прямокутних трикутників за катетом і бісектрисою, проведеною з вершини прямого кута.

Для цієї задачі в базі знань створюється два трикутники, які будуть порівнюватись.

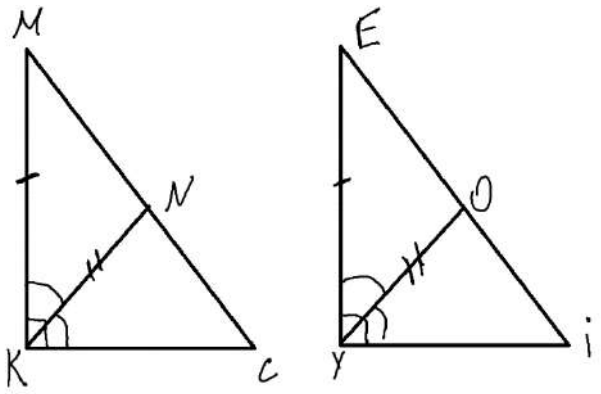


Рисунок 3.12 Задача №3

Алгоритм розв'язання:

1) Створити правила, що визначають рівні сторони та кути.
2) Бісектриса ділить кут навпіл, тож $\angle MKN = \angle EYO$.
3) Бісектриса поділяє трикутник на два трикутники.
4) За першою ознакою рівності трикутників трикутники MKN = EYO.
5) Оскільки утворені трикутники рівні, то всі елементи в них також рівні, отже $\angle M = \angle E$.
6) Трикутники MKC та EYI рівні за другою ознакою рівності трикутників.

SWRL правила, що необхідні для розв'язання даної задачі:

- Rule: Відрізок(?g), Відрізок(?k), має_довжину(?g, ?d1), має_довжину(?k, ?d2), subtract(0, ?d1, ?d2) -> рівні(?g, ?k) – Визначення рівних відрізків.

- Rule: Кут(?g), Кут(?k), має_градусну_міру(?g, ?d1), має_градусну_міру(?k, ?d2), subtract(0, ?d1, ?d2) -> рівні(?g, ?k) – Визначення рівних кутів.

- Rule: Кут(?k), Кут(?f), має_бісектрису(?k, ?b), належить(?b, ?f), має_градусну_міру(?k, ?c), divide(?res, ?c, 2) -> має_градусну_міру(?f, ?res) – Правило, що бісектриса ділить кут з якого проведена на два рівні кути.

- Rule: Трикутник(?t), Трикутник(?r), сторона_a(?t, ?k), сторона_a(?r, ?l), рівні(?k, ?l), сторона_b(?t, ?g), сторона_b(?r, ?p), рівні(?g, ?p), кут_α(?t, ?k1), кут_α(?r, ?k2), рівні(?k1, ?k2) -> рівні(?t, ?r) – Перша ознака рівності трикутників.

- Rule: Трикутник(?t), Трикутник(?r), рівні(?t, ?r), кут_β(?t, ?k3), кут_β(?r, ?k4) -> рівні(?k3, ?k4) – Якщо трикутники рівні, то відповідні кути теж рівні.

- Rule: Трикутник(?t), Трикутник(?r), сторона_a(?t, ?k), сторона_a(?r, ?l), рівні(?k, ?l), кут_α(?t, ?k1), кут_α(?r, ?k2), рівні(?k1, ?k2), кут_β(?t, ?k3), кут_β(?r, ?k4), рівні(?k3, ?k4) -> рівні(?t, ?r) – Друга ознака рівності трикутників.

Відповідь:

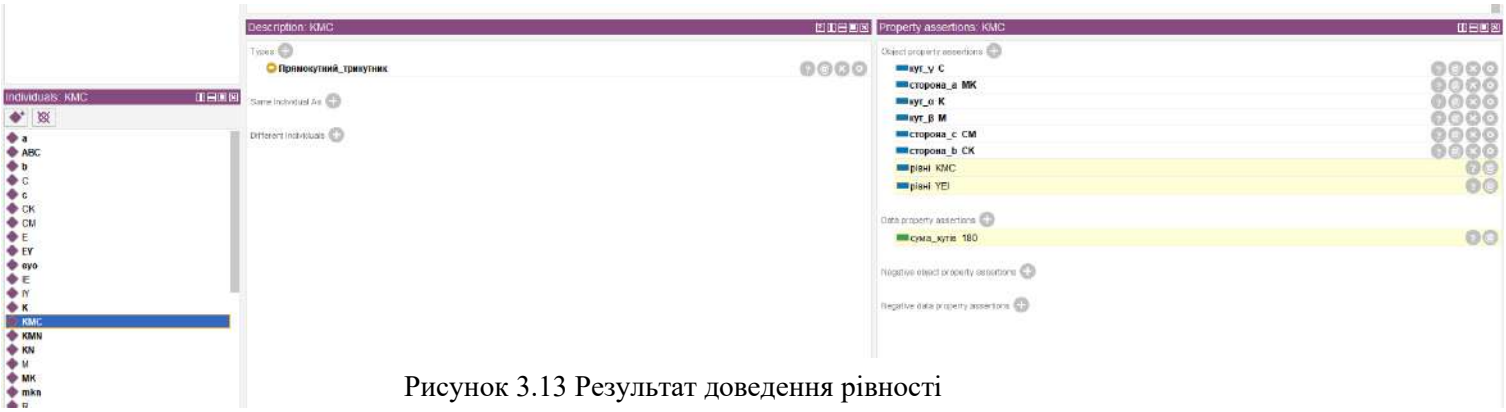


Рисунок 3.13 Результат доведення рівності

Під час розв’язання цього завдання та створення відповідних правил про рівність, виникла проблема. Кожен елемент вважає себе рівним собі, це сталось, оскільки в OWL немає припущення про унікальні імена. Існує функція

owl:differentFrom, що допоможе уникнути цієї проблеми, проте в даній версії Protégé вона не доступна.

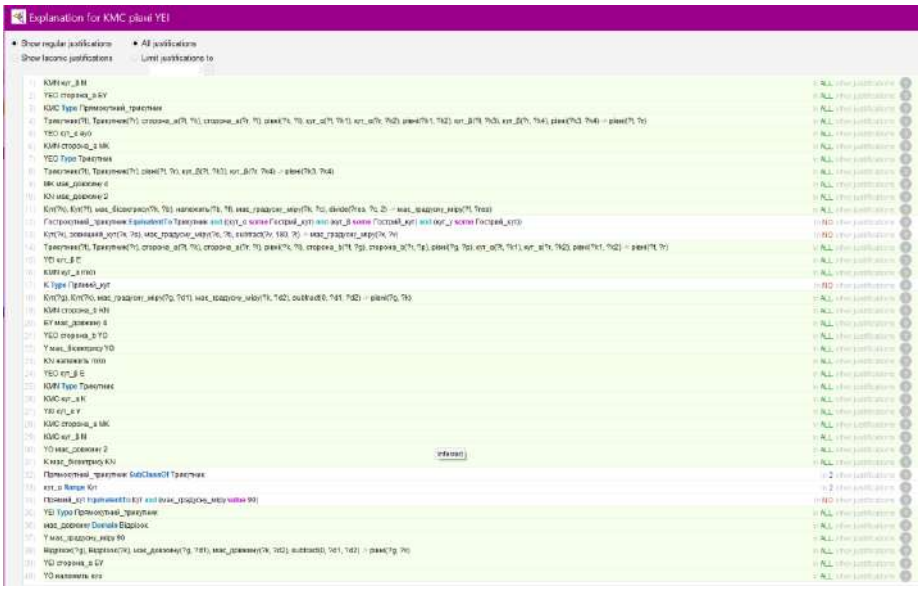


Рисунок 3.14 Пояснення до розв’язку третьої задачі

Розділ 4. Розробка експертної системи

4.1 Отримання результатів та кроків розв’язання

Для розробки експертної системи критично важливо мати можливість отримувати як результати виведення, так і кроки розв'язання, що призвели до цих результатів. Це дозволяє не лише отримати кінцеві висновки, але й зрозуміти логічний процес, за допомогою якого система до них дійшла.

Одним з основних інструментів для роботи з онтологіями є Protégé, який дозволяє інтегрувати різонери, такі як HermiT чи Pellet. Різонери виконують автоматичне виведення нових аксіом на основі існуючих даних в онтології. Важливо, щоб експертна система могла витягувати ці нові аксіоми та кроки, які призвели до їх утворення.

Кроки розв’язання включають в себе послідовність логічних операцій, що виконуються різонером для досягнення висновку. Наприклад, якщо різонер встановив, що певний індивідуум належить до певного класу, система повинна мати можливість відобразити, які саме аксіоми і правила були використані для цього висновку.

Використовуючи OWL API, вийшло програмно отримувати властивості індивідуумів, їхні класи, відносини між ними, а також нові аксіоми, згенеровані різонером. Це забезпечує можливість детального аналізу та подальшої обробки отриманої інформації.

OWL API — це API Java та довідкова реалізація для створення, обробки та серіалізації онтологій OWL. Це набір багатофункціональних інтерфейсів, які дозволяють гнучко маніпулювати онтологіями. Багато проектів використовують OWL-API як інструмент розробки. Приклади включають Protégé-4 , Swoop і OntoTrack. Остання версія API орієнтована на OWL 2.

Створено клас «OWLCommunicator», що реалізує спілкування з онтологіями в мові OWL.

Атрибути:

- ontologyFile та ontologyNamespace. Це шляхи або URI, які вказують на файл онтології та її простір імен відповідно.
- ontologyManager. Керує онтологією OWL.
- ontology. Представляє завантажену онтологію.
- ontologyDataFactory. Надає методи для створення сутностей OWL, таких як класи, властивості та індивідууми.

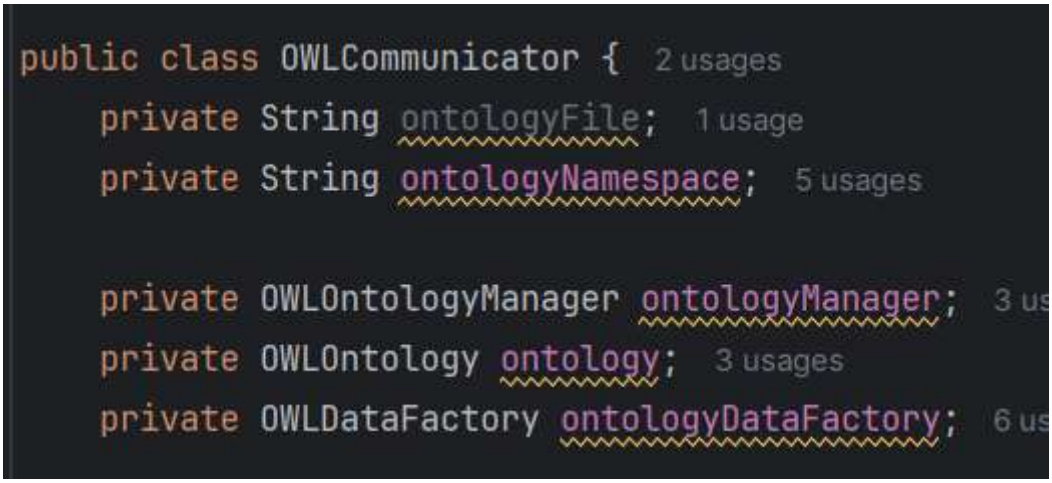


Рисунок 4.1 Атрибути класу «OWLCommunicator»

Метод «getDataPropertyLiteralValue» повертає літеральне значення вказаної властивості даних, пов'язане з певним індивідом. Використовує OWL-резонер (конкретно, екземпляр OpenlletReasoner), щоб запитати онтологію про значення властивостей даних.

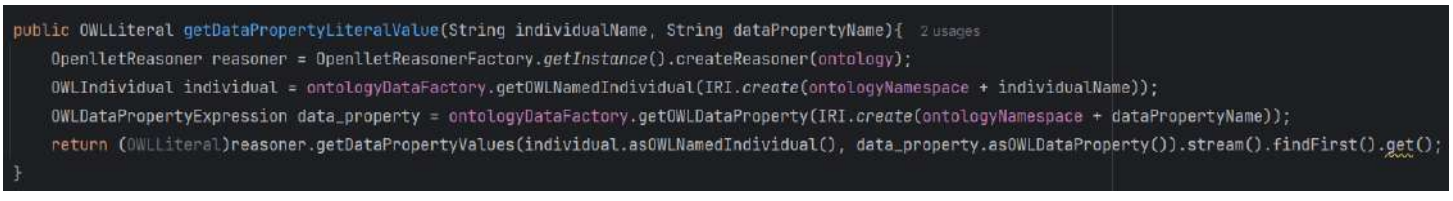


Рисунок 4.2 Метод «getDataPropertyLiteralValue»

«getDataPropertyStringValue» метод, який повертає рядкове значення властивості даних, пов'язане з певним індивідом. Він викликає getDataPropertyLiteralValue та витягує літеральне значення.

```
public String getDataPropertyStringValue(String individualName, String dataPropertyName){  1 usage
    return getDataPropertyLiteralValue(individualName, dataPropertyName).getLiteral();
}
```

Рисунок 4.3 Метод «getDataPropertyStringValue»

Метод «getExplanations» повертає пояснення того, чому певне значення властивості даних вважається наслідком для заданого індивідуума. Використовує OWL-резонер (OpenlletReasoner) та генератор пояснень Pellet (PelletExplanation) для створення пояснень. Пояснення повертаються у вигляді двовимірного масиву рядків, де кожен рядок представляє собою набір аксіом, що пояснюють наслідок.

```
public String[][] getExplanations(String individualName, String dataPropertyName) throws OWLException, IOException {  1 usage
    OpenlletReasoner reasoner = OpenlletReasonerFactory.getInstance().createReasoner(ontology);
    PelletExplanation explanation_generator = new PelletExplanation(reasoner);

    OWLIndividual individual = ontologyDataFactory.getOWLNamedIndividual(IRI.create(ontologyNamespace + individualName));
    OWLDataPropertyExpression data_property = ontologyDataFactory.getOWLDataProperty(IRI.create(ontologyNamespace + dataPropertyName));

    OWLLiteral literal = getDataPropertyLiteralValue(individualName, dataPropertyName);

    OWLAxiom assert_axiom = ontologyDataFactory.getOWLDataPropertyAssertionAxiom(data_property, individual, literal);

    Set<Set<OWLAxiom>> explanations = explanation_generator.getEntailmentExplanations(assert_axiom);

    ArrayList<String[]> res = new ArrayList<>();
    for(Set<OWLAxiom> exp : explanations){
        ArrayList<String> axioms = new ArrayList<>();
        for(OWLAxiom axiom : exp){
            StringWriter out = new StringWriter();
            TextBlockWriter writer = new TextBlockWriter(out);
            ManchesterSyntaxObjectRenderer renderer = new ManchesterSyntaxObjectRenderer(writer);
            axiom.accept(renderer);
            writer.flush();
            out.flush();
            axioms.add(out.toString().replaceAll( regex: "[\\n\\r]", replacement: "").replaceAll( regex: "\\s{2,}", replacement: " ").strip());
            writer.close();
            out.close();
        }

        res.add(axioms.toArray(new String[0]));
    }
```

Рисунок 4.4 Метод «getExplanations»

Оскільки інтерфейс та в цілому експертна система розроблена на мові Python, наступним кроком було зрозуміти, яким чином доступатись до нових

виведених аксіом та кроків з Python-програми.

Було розглянуто два варіанти для вирішення цього питання: зберігати виведені дані у файл і зчитувати їх у Python-програмі, використовувати JAR-файл та бібліотеку Py4J для доступу до виведених аксіом.

Тож для доступу до даних було обрано другий варіант, оскільки замість того, щоб зберігати аксіоми у файлі та потім їх зчитувати, використання JAR-файлу дозволяє безпосередньо викликати методи Java-коду з Python, що забезпечує доступ до всіх можливостей класів і методів Java, включаючи логіку обробки даних, що може бути складно або неможливо реалізувати через файл. Прямий виклик Java-методів з Python через Py4J зазвичай швидший за зчитування даних з файлу, особливо при роботі з великими обсягами даних або складними операціями. Також Java має багатий набір бібліотек для роботи з онтологіями та різонерами (наприклад, OWL API, Openllet). Використання Py4J дозволяє Python-програмам скористатися цими бібліотеками безпосередньо, що спрощує розробку та забезпечує доступ до потужних інструментів.

Отже код на Python використовує бібліотеку Py4J для взаємодії з Java-класом OWLCommunicator, який міститься в JAR-файлі Assist.jar.

```
from py4j.java_gateway import JavaGateway

gg = JavaGateway.launch_gateway(classpath="Assist.jar")

owlComm = gg.jvm.org.example.OWLCommunicator("file:ontology.owl","http://www.semanticweb.org/nastia/ontologies/math_assistent#")

expls = owlComm.getExplanations("R","has_length")
```

Рисунок 4.5 Отримання виведених значень

- JavaGateway.launch_gateway(classpath="Assist.jar"). Запускає Java-шлюз і завантажує класовий шлях, вказуючи на Assist.jar, що містить Java-класи.
- gg.jvm.org.example.OWLCommunicator("file:ontology.owl","http://www.semanticweb.org/nastia/ontologies/math_assistent#"). Створює екземпляр класу OWLCommunicator з вказаним файлом онтології та простором імен.

- owlComm.getExplanations("R","has_length"). Викликає метод Java-класу для отримання пояснень щодо певного індивіда та властивості, повертаючи їх у Python-змінну expls.

Цей підхід дозволяє безпосередньо викликати методи Java-класів з Python, що спрощує інтеграцію логіки та даних між двома мовами програмування.

4.2 Створення експертної системи

У даному підрозділі буде розглянуто логіку створеної системи. Даний код реалізує систему підтримки користувача в рішенні завдань з використанням онтологій. Основна мета цієї системи — допомогти користувачеві проходити через послідовність кроків для вирішення завдання, забезпечуючи зворотний зв'язок та перевірку правильності кожного кроку.

Окремий опис кожного компоненту:

Клас Assistant є центральним компонентом системи, що керує процесом вирішення завдання. Його основні методи та атрибути включають:

- __init__(self, task, ontology_filename). Конструктор, який ініціалізує об'єкт Assistant. Він приймає завдання (task) і файл онтології (ontology_filename), створюючи екземпляр OWLCommunicator для взаємодії з онтологією. Завантажуються доступні кроки завдання і визначаються кроки, необхідні для вирішення завдання.

```
class Assistant:
    def __init__(self, task, ontology_filename):
        self.task = task
        self.communicator = OWLCommunicator(ontology_filename, task.owl_namespace)
        self.available_steps = task.steps
        self.solution_steps = []
        self.reasoner_steps = self.get_reasoner_steps()
```

Рисунок 4.6 Конструктор

- get_reasoner_steps(self). Метод отримує пояснення від онтологічного модуля (communicator) для визначення кроків, необхідних для досягнення кінцевого результату. Він фільтрує ці пояснення, знаходить відповідні кроки серед доступних і перевіряє їх коректність.

```
def get_reasoner_steps(self):
    explanations = self.communicator.get_explanations(self.task.result_individual_name, self.task.result_data_property_name)
    explanations = filter(lambda x: x.startswith("Rule("),explanations[0])
    res = []
    for expl in explanations:
        steps = list(filter(lambda x: expl.replace(",","").replace(" ","") == x.owl_explanation_pattern.replace(",","").replace(" ",""), self.available_steps))
        if len(steps) == 1:
            step = copy.deepcopy(steps[0])
            step.correct_result_value = self.communicator.get_value(step.result_individual_name, step.result_data_property_name)
            res.append(step)
    if len(res) == 0:
        raise Exception("Not all steps found in explanations")

    return res
```

Рисунок 4.7 Метод get_reasoner_steps

- is_allowed_next_step(self, step). Метод перевіряє, чи дозволено виконання наступного кроку на основі визначеної послідовності кроків (reasoner_steps).

```
def is_allowed_next_step(self, step):
    return True if step.id == self.reasoner_steps[len(self.solution_steps)].id else False
```

Рисунок 4.8 Метод is_allowed_next_step

- is_valid_next_step(self, step). Метод перевіряє, чи введене користувачем значення є правильним, порівнюючи його з очікуваним значенням. Перевіряється, чи крок дозволений, і чи введене значення близьке до очікуваного з допустимою похибкою.

```
def is_valid_next_step(self, step):
    if step.entered_result_value.replace('.','',1).isdigit():
        float_entered = float(step.entered_result_value)
        float_correct = float(self.reasoner_steps[len(self.solution_steps)].correct_result_value)
        vals_equal = True if abs(float_entered - float_correct) < abs(float_correct / 10000.0) else False
        return True if self.is_allowed_next_step(step) and vals_equal else False
    else:
        return False
```

Рисунок 4.9 Метод is_valid_next_step

- is_task_solved(self). Метод визначає, чи було завдання вирішено, перевіряючи, чи відповідає останній крок очікуваному результату.

```
def is_task_solved(self):
    if (len(self.solution_steps) > 0 and self.solution_steps[-1].result_individual_name == self.task.result_individual_name and
        self.solution_steps[-1].result_data_property_name == self.task.result_data_property_name): return True
    else: return False
```

Рисунок 4.10 Метод is_task_solved

Клас OWLCommunicator забезпечує взаємодію з онтологічним модулем, що базується на OWL (Web Ontology Language). Основні методи цього класу:

- __init__(self, ontology_filename, owl_namespace). Конструктор, який ініціалізує з'єднання з онтологією через Java Gateway, використовуючи файл онтології та простір імен OWL.

```
class OWLCommunicator:
    def __init__(self, ontology_filename, owl_namespace):
        self.ontology_filename = ontology_filename
        self.owl_namespace = owl_namespace
        self.gg = JavaGateway.launch_gateway(classpath="Assist.jar")
        self.javaComm = self.gg.jvm.org.example.OWLCommunicator(f"file:{ontology_filename}", owl_namespace)
```

Рисунок 4.11 Конструктор

- get_explanations(self, individual_name, data_property_name). Метод отримує пояснення для певного індивіда та властивості даних з онтології.

```
def get_explanations(self, individual_name, data_property_name):
    return self.javaComm.getExplanations(individual_name,data_property_name)
```

Рисунок 4.12 get_explanations

- get_value(self, individual_name, data_property_name). Метод отримує значення певної властивості даних для індивіда з онтології.

```
def get_value(self, individual_name, data_property_name):
    return self.javaComm.getDataPropertyStringValue(individual_name,data_property_name)
```

Рисунок 4.13 get_value

Клас Task представляє завдання, яке користувач повинен вирішити. Основні методи та атрибути включають:

- __init__(self, task_id). Конструктор, який ініціалізує об'єкт завдання з унікальним ідентифікатором.

```
class Task:
    def __init__(self, task_id):
        self.id = task_id
        self.name = ""
        self.title = ""
        self.description = ""
        self.result_individual_name = ""
        self.result_data_property_name = ""
        self.owl_namespace = ""
        self.figure_file_name = ""
        self.steps = []
```

Рисунок 4.14 Конструктор

- create_from_hash(task_id, figure_file_name, h). Статичний метод, який створює завдання з хешу даних. Метод заповнює поля завдання, такі як ім'я, опис, назва, кінцевий індивід і властивість даних, простір імен OWL та кроки завдання.

```
@staticmethod
def create_from_hash(task_id, figure_file_name, h):
    task = Task(task_id)
    task.name = h["name"]
    task.title = h["title"]
    task.description = h["description"]
    task.result_individual_name = h["result_individual_name"]
    task.result_data_property_name = h["result_data_property_name"]
    task.owl_namespace = h["owl_namespace"]
    task.figure_file_name = figure_file_name
    task.steps = list(map(lambda x: Step.create_from_hash(x["id"], x) , h["steps"]))
    return task
```

Рисунок 4.15 create_from_hash

Клас Step представляє окремий крок у процесі вирішення завдання. Основні методи та атрибути включають:

- __init__(self, step_id). Конструктор, який ініціалізує об'єкт кроку з унікальним ідентифікатором.

```
class Step:
    def __init__(self, step_id):
        self.id = step_id
        self.name = ""
        self.title = ""
        self.description = ""
        self.result_individual_name = ""
        self.result_data_property_name = ""
        self.correct_result_value = ""
        self.entered_result_value = ""
        self.owl_explanation_pattern = ""
```

Рисунок 4.16 Конструктор

- create_from_hash(step_id, h). Статичний метод, який створює крок з хешу даних. Метод заповнює поля кроку, такі як назва, опис, індивід і властивість даних, очікуване значення результату та шаблон пояснення OWL.

```
@staticmethod
def create_from_hash(step_id, h):
    step = Step(step_id)
    step.name = h["name"]
    step.title = h["title"]
    step.description = h["description"]
    step.result_individual_name = h["result_individual_name"]
    step.result_data_property_name = h["result_data_property_name"]
    step.correct_result_value = h["correct_result_value"]
    step.owl_explanation_pattern = h["owl_explanation_pattern"]
    return step
```

Рисунок 4.17 create_from_hash

Клас Gateway відповідає за завантаження та створення завдань з файлової системи. Основні методи включають:

- load_tasks(self). Метод завантажує всі доступні завдання з директорії tasks, повертаючи список об'єктів Task.

```
def load_tasks(self):
    return list(map(lambda x: self.create_task_from_dir(x), [os.path.basename(os.path.normpath(f)) for f in (Path('.') / "tasks").glob("*") if f.is_dir()]))
```

Рисунок 4.18 load_tasks

- create_task_from_dir(self, dir_name). Метод створює окреме завдання з вказаної директорії, зчитуючи JSON та PNG файли, які містять дані завдання.

```
def create_task_from_dir(self, dir_name):
    json_files = [f for f in (Path('.') / "tasks" / dir_name).glob("*.json") if f.is_file()]
    if len(json_files) == 0 or len(json_files) > 1: raise Exception(f"One json file expected in task directory: {dir_name}")
    figure_files = [f for f in (Path('.') / "tasks" / dir_name).glob("*.png") if f.is_file()]
    if len(figure_files) == 0 or len(figure_files) > 1: raise Exception(f"One png file expected in task directory: {dir_name}")
    with open(json_files[0], 'r', encoding='utf8') as json_file:
        task_json = json.load(json_file)
        return Task.create_from_hash(dir_name, figure_files[0], task_json)
```

Рисунок 4.19 create_task_from_dir

Усі завдання зберігаються у JSON-файлі task. Формат задання завдання наступний:

```
{
  "name" : "Завдання 01",
  "title" : "Знаходження радіуса вписаного кола",
  "description" : "Знайти радіус вписаного кола в прямокутний трикутник зі сторонами 15, 20, 25. ",
  "result_individual_name" : "r",
  "result_data_property_name" : "has_length",
  "owl_namespace" : "http://www.semanticweb.org/nastia/ontologies/math_assistent#",
  "steps" : [
    {
      "id" : "3",
      "name" : "Радіус",
      "title" : "Радіус вписаного кола",
      "description" : "Знайти радіус вписаного кола в трикутник як частку площі та півпериметра",
      "result_individual_name" : "r",
      "result_data_property_name" : "has_length",
      "correct_result_value" : "5",
      "owl_explanation_pattern" : "Rule(Triangle(?t), perimeter(?t, ?P), area(?t, ?s), divide(?p, ?P, 2), divide(?res, ?s, ?p),
    },
    {
      "id" : "1",
      "name" : "Периметр",
      "title" : "Периметр трикутника",
      "description" : "Знайти периметр трикутника як суму довжин всіх його сторін",
      "result_individual_name" : "ABC",
      "result_data_property_name" : "perimeter",
      "correct_result_value" : "60",
      "owl_explanation_pattern" : "Rule(Triangle(?t), side_a(?t, ?s), side_b(?t, ?x), side_c(?t, ?g), has_length(?s, ?l1), has_l
    },
    {
```

Рисунок 4.20 Task

Для кожного завдання зберігається умова, ім’я індивідуала, що необхідно знайти та значення заданої числової властивості.

Кожен крок складається з опису, ім’я індивідуала, числової властивості,

очікуваного результату та шаблон пояснення OWL(кроків, що були виведені різонером).

4.3 Створення експертної системи

Дана система демонструє застосування онтологій для моделювання та вирішення геометричних задач. Кожен крок детально описує необхідні дії та очікувані результати, що допомагає користувачеві проходити через завдання крок за кроком, забезпечуючи правильність кожного етапу.

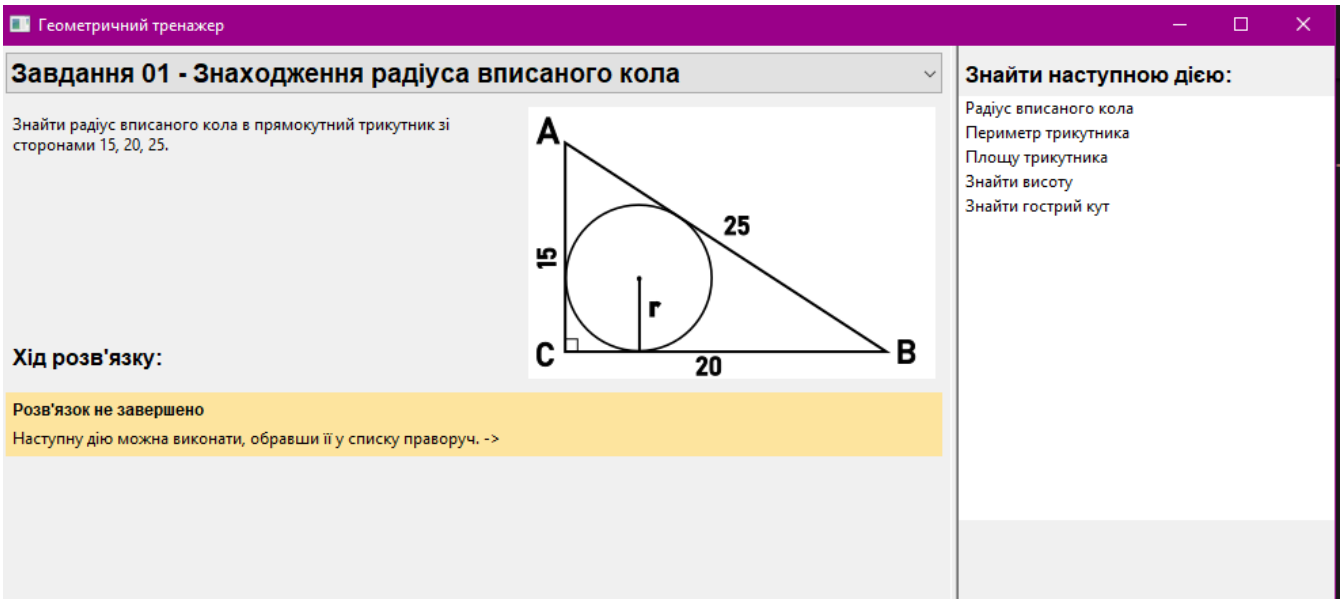


Рисунок 4.21 Початок розв’язання задачі

Користувачу надається умова та малюнок до задачі. Праворуч перемішані кроки, що знадобляться для розв’язання цієї задачі. Не всі кроки мають бути використаними, оскільки серед правильних кроків наявні зайві.

Щоб почати розв’язання потрібно натиснути на крок. Якщо на даному етапі

цей крок не передбачається, буде виведено повідомлення про це.

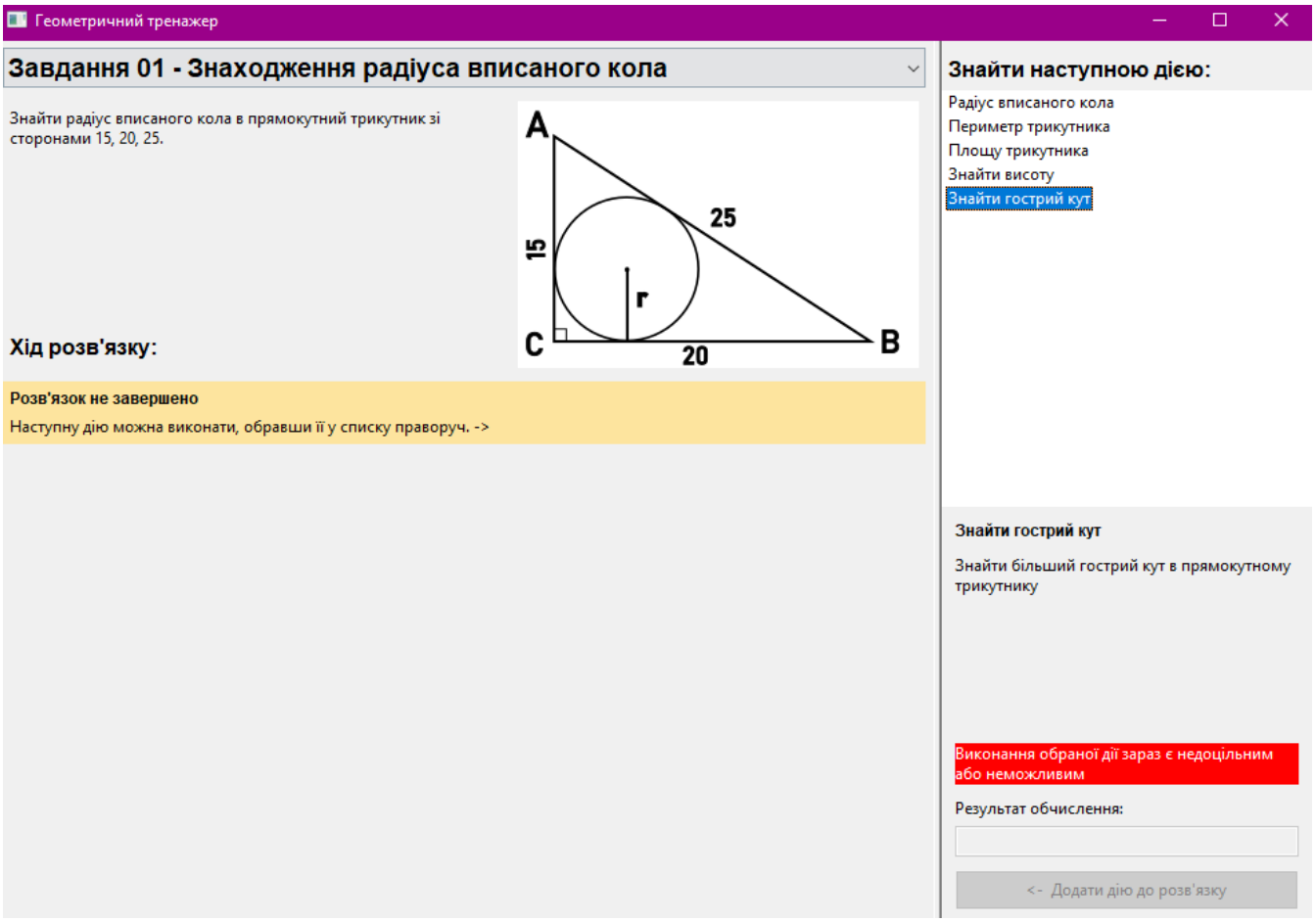


Рисунок 4.22 Неправильно визначений наступний крок

Якщо користувач правильно обрав крок, то надається можливість ввести значення, що буде відповіддю для шуканого елементу на даному етапі.

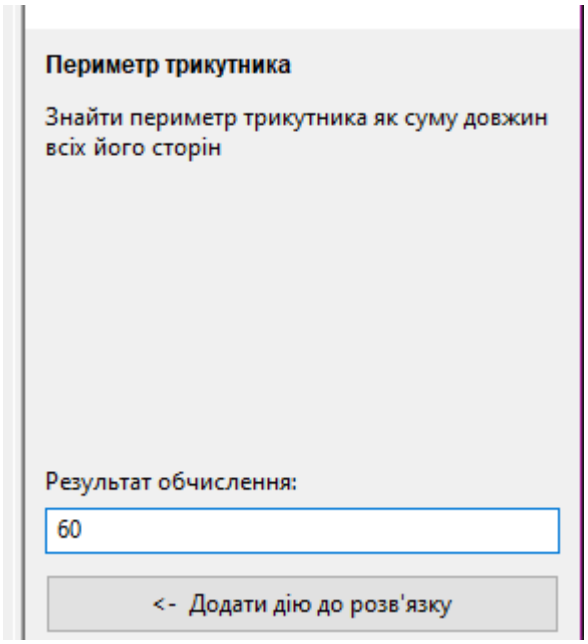
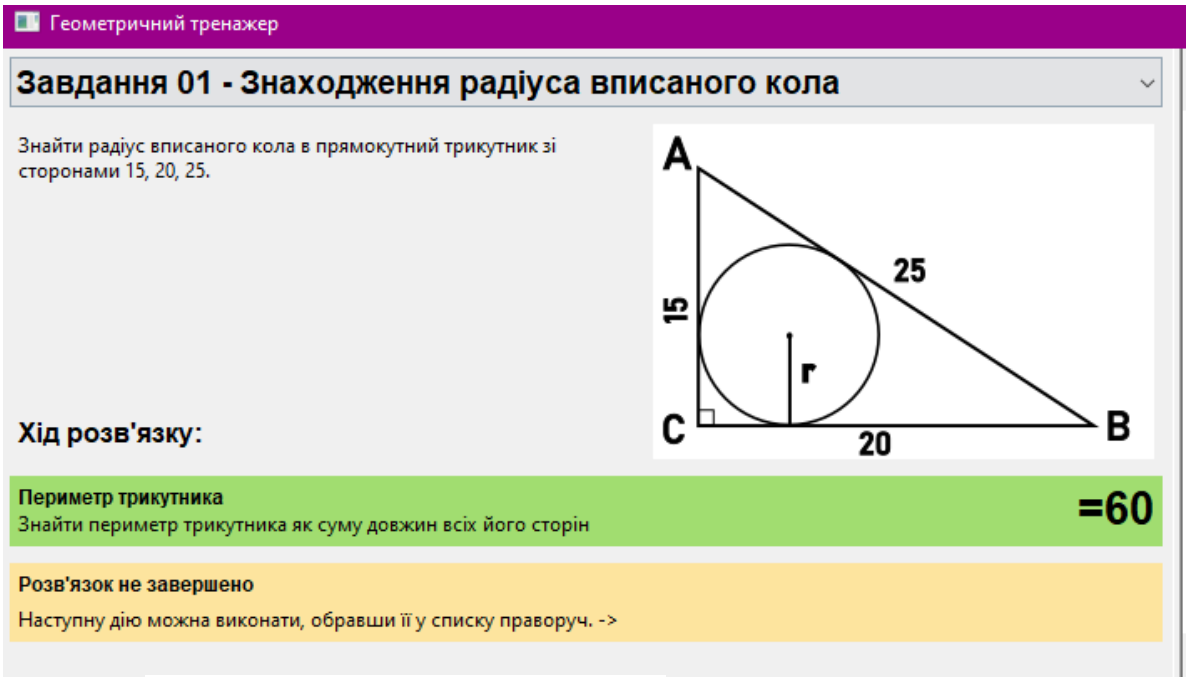


Рисунок 4.23 Введення шуканого значення

Якщо введена відповідь правильна, то розв’язок зберігається у вікні ліворуч.



Якщо результат було введено некоректно, то з’являється відповідне повідомлення та кнопка «Додати дію до розв’язку» стає неактивною.

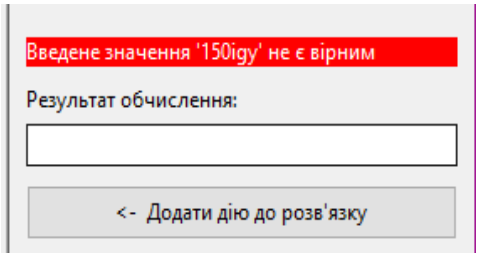


Рисунок 4.25 Некоректно введене значення

У результаті користувач має змогу поступово правильно розв’язувати завдання та зберігати хід розв’язку.

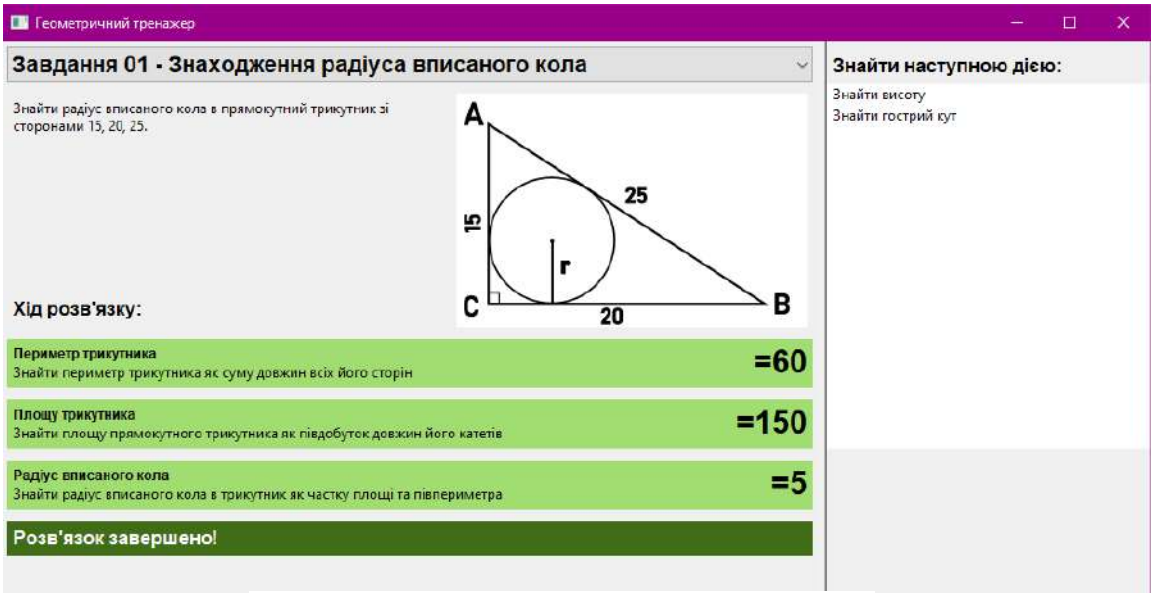


Рисунок 4.26 Результат розв’язання задачі

Висновки

Отже, реалізація онтології в Protege та створення бази знань є важливою складовою для автоматизації процесів, організації, управління та використання знань у різних сферах, зокрема в геометрії. У ході виконання цієї кваліфікаційної роботи було досліджено можливості використання Protege в контексті геометричних задач. Було розроблено базу знань у Protege, метою якої є вирішення конкретних простих геометричних задач.

Аналіз фінальних результатів дослідження показав, що розроблена база знань є ефективною для розв’язання геометричних задач. Реалізована в Protege база знань дозволяє структурувати та організувати геометричні знання, роблячи їх доступнішими для користувачів. Для ефективного аналізу інформації використовувалися правила SWRL.

Створена система має значне значення як для окремих користувачів, так і для освітнього процесу загалом. База знань може стати потужним інструментом для покращення процесу навчання та розвитку навичок учнів у вирішенні геометричних задач. Використання структурованих знань у базі знань дозволяє учням глибше осягнути геометричні концепції та покращити свої вміння у вирішенні завдань.

Таким чином, впровадження онтології та розробка бази знань є перспективним напрямом, який може значно покращити якість освіти, забезпечуючи учням інструменти для ефективного та глибокого розуміння предмету.

Список використаних джерел

1. protégé. URL: https://protege.stanford.edu/conference/2007/slides/08.01_OConnor.pdf
2. OWL 2 and SWRL Tutorial. Index of /. URL: https://dior.ics.muni.cz/~makub/owl/
3. OWL - Semantic Web Standards. W3C. URL: https://www.w3.org/OWL/
4. Semantic SWRL rule. https://grimstad.uia.no/janpn/IKT437/2016/slides/pdf/rules_2016.pdf
5. Swrl rule documentation. https://www.w3.org/Submission/SWRL/
6. Owl 101. Cambridge Semantics. URL: https://cambridgesemantics.com/blog/semantic-university/learn-owl-rdfs/owl-101/
7. Смиш О. ОРГАНІЗАЦІЯ ІНТЕРФЕЙСУ ДЛЯ ВЗАЄМОДІЇ З КОРИСТУВАЧЕМ В РЕКОМЕНДАЦІЙНІЙ СИСТЕМІ. Наука і техніка сьогодні. 2024. № 3(31). URL: https://doi.org/10.52058/2786-6025-2024-3(31)-980-989
8. GitHub - owlcs/owlapi: OWL API main repository. GitHub. URL: https://github.com/owlcs/owlapi