Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студентки *Рибки Вероніки Олександрівни*
(ПІБ)

академічної групи *122М-20-2*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

на тему: *Методи, алгоритми та інформаційна технологія розпізнавання неістинності висловлювань*

____________________ *В.О. Рибка*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	Проф. Мещеряков Л.І.			
економічний	Доц. Касьяненко Л.В.			

Рецензент				

Нормоконтролер	Доц. Реута О.В.			

Дніпро
2022

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

______________ І.М. Удовик
(підпис) (прізвище, ініціали)

« » ____________ 20 ____ Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122М-20-2* *Рибці Вероніці Олександрівні*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Методи, алгоритми та інформаційна технологія розпізнавання неістинності висловлювань*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 10.12.2021 р. № 1036-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес роботи програмної реалізації методу перевірки виводимості формул Куайна – Мак-Класкі на мові програмування C#.

Предмет досліджень – програмні методи вирішення проблеми нестачі пам'яті при реалізації методу Куайна – Мак-Класкі на мові програмування C#.

Мета роботи – оптимізація роботи програмної реалізації методу Куайна – Мак-Класкі при нестачі оперативної пам'яті, використовуючи існуючі програмні методи мови C#.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна результатів роботи полягає в дослідженні можливих способів оптимізації роботи алгоритму Куайна - Мак-Класкі на платформі .NET.

Практична цінність полягає у тому, що результати роботи, отримані в ході дослідження, можуть застосовуватися як в розробці додатків, що використовують в собі алгоритм Куайна – Мак-Класкі, а також в практиці викладання дисциплін, пов’язаних з численням висловлень.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	Строки виконання робіт (початок – кінець)
Аналіз теми та постановка задачі	30.09.2021-12.10.2021
Аналіз предметної області, збір інформації.	13.10.2021-31.10.2021
Практична реалізація методів та шляхів вирішення проблеми. Аналіз результатів.	01.11.2021-16.12.2021
Написання пояснювальної записки.	17.12.2021 – 10.01.2022

Завдання видав ______ Мещеряков Л.І.

(підпис) (прізвище, ініціали)

Завдання прийняв до виконання ______ Рибка В.О. _

(підпис) (прізвище, ініціали)

Дата видачі завдання: 12.09.2022 р.

Термін подання дипломного проекту до ЕК 17.12.2022р.

РЕФЕРАТ

Пояснювальна записка: 103 стор., 22 рис., 5 таблиць, 3 додатки, 51 джерело.

Об'єкт дослідження: процес роботи програмної реалізації методу перевірки виводимості формул Куайна – Мак-Класкі на мові програмування C#.

Предмет дослідження: програмні методи вирішення проблеми нестачі пам’яті при реалізації методу Куайна – Мак-Класкі на мові програмування C#.

Мета магістерської роботи: оптимізація роботи програмної реалізації методу Куайна – Мак-Класкі при нестачі оперативної пам’яті, використовуючи існуючі програмних методів мови C#.

Методи дослідження. Для виконання поставлених завдань були використані методи алгебри логіки, методи числення висловлень, методи перевірки виводимості логічних формул, а також методи покращення роботи програми при нестачі пам’яті.

Наукова новизна результатів роботи полягає в дослідженi можливих способів оптимізації роботи алгоритму Куайна - Мак-Класкі на платформі .NET.

Практична цінність результатів полягає в тому, що результати роботи, отримані в ході дослідження, можуть застосовуватися як в розробці додатків, що використовують в собі алгоритм Куайна – Мак-Класкі, а також в практиці викладання дисциплін, пов’язаних з численням висловлень.

У розділі «Економіка» проведено розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПО і тривалості його розробки.

Список ключових слів: істинність, розпізнавання істинності міркувань, обчислювальна система, штучний інтелект, числення висловлень, метод Куайна – Мак-Класкі, .NET, C#.

ABSTRACT

Explanatory note: 103 pages, 22 figures, 5 tables, 3 applications, 51 sources.

Object of research: the process of software implementation of the Quine – McCluskey’s method of checking the derivability of formulas in the C# programming language.

Subject of research: software methods to solve the problems of lack of RAM in the implementation of the Quine-McCluskey’s method in the C # programming language.

Purpose of Master's thesis: optimization of the software implementation of the Quine-McCluskey method in the absence of RAM, using existing C# programming methods.

Research methods. Methods of algebra of logic, methods of calculus of statements, methods of checking the derivability of logical formulas, as well as methods of improving the program in the absence of memory were used to perform the tasks.

Originality of research lies in the study of possible ways to optimize the operation of the Quine - McCluskey algorithm in.NET.

Practical value of the results. The results of the research can be used both in the development of applications that use the Quine-McCluskey algorithm, as well as in the practice of teaching disciplines.

In the Economics section we calculated the complexity of software development and the costs of software development, the duration of the actual development are calculated.

Keywords: truthfulness, reasoning recognition, computer system, artificial intelligence, calculus, Quine-McCluskey method, .NET, C#.

ВСТУП... 9
РОЗДІЛ 1. Логічні функції як складова частина побудови систем штучного інтелекту 11
1.1 Загальні відомості про штучний інтелект.. 11
1.2 Штучний інтелект та логічне програмування 14
1.3 Числення висловлень 15
1.3.1 Мова логіки висловлень 16
1.3.2 Семантика числення висловлень 18
1.3.3 Тотожні формули (тавтології) в численні висловлень 21
1.3.4 Основні та похідні форми аргументів 21
1.3.5 Основні проблеми числення висловлень 24
1.4 Числення предикатів 26
1.4.1 Аксіоматика числення предикатів 28
1.4.1 Виведення формул і теорем 30
1.4.2 Властивості числення предикатів 30
1.5 Методи та алгоритми перевірки виводимості формул 31
1.6 Метод перевірки виводимості формул Куайна – Мак-Класкі 33
1.6.1 Метод Куайна. Загальні положення 33
1.6.2 Метод Куайна - Мак-Класкі. Загальні положення 39
Висновки до першого розділу 41
РОЗДІЛ 2. Оптимізація роботи методу Куайна - Мак-Класкі 44
2.1 Проблеми пов’язані з реалізацією методу в середовищі .NET та можливі варіанти їх вирішення 44
2.2.1 Визначення вузлів троїчного дерева як структур 45
2.2.2 Додавання пулу об’єктів 48
2.2.3 Явний виклик методів Garbage Collector 50
Висновки до другого розділу 54
РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ 55

3.1 Програмна реалізація методу Куайна - Мак-Класкі ... 55

3.2 Застосування запропонованих методів вирішення проблеми реалізації методу Куайна - Мак-Класкі на платформі .NET ... 58

3.2.1 Результати заміни класів вузлів троїчного дерева на структури ... 58

3.2.2 Результати додавання пулу об’єктів до програми ... 59

3.2.3 Результати явного виклику методів Garbage Collector ... 60

Висновки до третього розділу ... 65

РОЗДІЛ 4. ЕКОНОМІКА ... 67

4.1. Визначення трудомісткості розробки програмного забезпечення ... 67

4.2. Розрахунок витрат на створення програмного забезпечення ... 70

4.3. Маркетингові дослідження ... 71

4.4. Економічна ефективність ... 71

ВИСНОВКИ ... 72

ПЕРЕЛІК ВИКОРИСТАННИХ ДЖЕРЕЛ ... 76

Додаток А. ЛІСТИНГ ПРОГРАМИ ... 81

Додаток Б. ВІДГУК КЕРІВНИКА ... 100

Додаток В. РЕЦЕНЗІЯ ... 102

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ ... 103

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ШІ – штучний інтелект;

НМ – нейрона мережа;

ЛП – логічне програмування;

ЛФ – логічна функція;

ДНФ – диз’юнктивна нормальна форма;

КНФ – кон’юктивна нормальна форма;

ДДНФ – досконала диз’юнктивна нормальна форма;

ДКНФ – досконала кон’юнктивна нормальна форма

ППФ – правильно побудовані формули;

МДНФ - мінімальна диз'юнктивна нормальна форма;

ЕОМ – електронна обчислювальна машина;

GC – Garbage Collector.

ВСТУП

Актуальність дослідження. На даному етапі розвитку інформаційних технологій, штучний інтелект має один з найвищих пріоритетів у розвитку. Саме він вважається майбутнім для усього всесвіту і тому багато уваги та досліджень присвячені саме найрізноманітнішим технологіям та інноваціям, спрямованим на розвиток даної сфери.

В основі своєї роботи штучний інтелект має методи та алгоритми алгебри логіки. Таким чином, навіть найменші наукові дослідження в цій сфері можуть привести до швидшого розвитку штучного інтелекту як технології майбутнього.

Серед багатьох різноманітних за призначенням методів числення висловлень існують методи перевірки виводимості логічних формул, які відіграють важливу роль при реалізації систем штучного інтелекту. Одним з таких методів є метод Куайна – Мак-Класкі. Дослідження спрямоване на дослідження можливих шляхів оптимізації цього метода при його програмній реалізації на мові програмування C#.

Мета дослідження оптимізація роботи програмної реалізації методу Куайна – Мак-Класкі при нестачі оперативної пам’яті, використовуючи існуючі програмних методів мови C#.

Завдання дослідження. Для досягнення поставленої мети в роботі сформульовані і вирішені такі завдання:

1. Викласти основні поняття алгебри логіки та числення висловлень.
2. Визначити основні методи перевірки виводимості формул.
3. Визначити основні проблеми, що виникають в програмній реалізації методу Куайна – Мак-Класкі.
4. Знайти можливі шляхи оптимізації програмної реалізації методу Куайна – Мак-Класкі.
5. Проаналізувати доцільність застосування методів оптимізації, а також результати їх застосування.

Об'єкт дослідження. процес роботи програмної реалізації методу перевірки виводимості формул Куайна – Мак-Класкі на мові програмування C#.

Предмет дослідження. програмні методи вирішення проблеми нестачі пам’яті при реалізації методу Куайна – Мак-Класкі на мові програмування C#.

Методи дослідження. Для виконання поставлених завдань були використані методи алгебри логіки, методи числення висловлень, методи перевірки виводимості логічних формул, а також методи покращення роботи програми при нестачі пам’яті.

Наукова новизна результатів роботи полягає в дослідженні можливих способів оптимізації роботи алгоритму Куайна - Мак-Класкі на платформі .NET.

Практичне значення результатів роботи полягає в тому, що результати роботи, отримані в ході дослідження, можуть застосовуватися як в розробці додатків, що використовують в собі алгоритм Куайна – Мак-Класкі, а також в практиці викладання дисциплін, пов’язаних з численням висловлень.

Особистий внесок автора: пошук та реалізація методів оптимізації роботи методу Куайна – Мак-Класкі на платформі .NET. Аналіз отриманих результатів.

Структура і обсяг роботи. Робота складається з вступу, чотирьох розділів і висновків. Містить 103 сторінки, в тому числі 60 сторінок тексту основної частини з 22 рисунками, списку використаних джерел з 51 найменуваннями на 5 сторінках, 3 додатка на 23 сторінках.

РОЗДІЛ 1.
ЛОГІЧНІ ФУНКЦІЇ ЯК СКЛАДОВА ЧАСТИНА ПОБУДОВИ СИСТЕМ ШТУЧНОГО ІНТЕЛЕКТУ

1.1. Загальні відомості про штучний інтелект

Штучним інтелектом (ШІ) називають вміння комп’ютерної системи обробляти, використовувати та покращувати здобуті знання та навички.

Алгоритм вирішення деякої задачі у більшості невідомий заздалегідь. У даної науки відсутнє точне її визначення, оскільки у філософії ще немає точного вирішення питання про статус інтелекту людини. Також відсутній і точний критерій, що дозволяє зрозуміти чи досяг комп'ютер «розумності». Проте існують декілька гіпотез серед яких, наприклад, тест Тюрінга або гіпотеза Ньюелла-Саймона.

Серед безлічі підходів до розробки систем штучного інтелекту виділяють два основні:

- Низхідний (семіотичний) — створення символьних систем, які моделюють такі психічні процеси як, наприклад, мислення, припущення, мову, почуття, творчість тощо;
- Висхідний (біологічний) — вивчення штучних нейронних мереж і еволюційні обчислення, що моделюють інтелектуальну поведінку на основі менших «неінтелектуальних» елементів.

Штучний інтелект як галузь був започаткований в 1956 року. На сьогоднішній день розвиток даної галузі перебуває на «підйомі» і спирається на застосування результатів, що були досягнуті в інших галузях науки.

З точки зору наукової дисципліни ШІ має такі основні напрямки:

- машинне мислення (охоплює процеси планування, представлення знань і міркування, пошук та оптимізацію);
- машинне навчання (умовно поділяється на глибоке навчання і навчання з підкріпленням);
- робототехніка (включає в себе управління, ситуаційне сприйняття, датчики і приводи, а також інтеграцію усіх інших методів в кібер-фізичні системи).

Можна виділити 4 основні, проте досить різні підходи до вивчення штучного інтелекту:

1. Логічний підхід. Основою цього підходу є алгебра логіки, яка отримала свій подальший розвиток у вигляді числення предикатів. В ньому вона розширена шляхом введення предметних символів та їх взаємовідносин. Окрім того, кожна така машина має блок генерації цілі, і система виводу намагається довести дану ціль як теорему. При досягнені цілі послідовність використаних правил дозволить отримати набір послідовних дій, які є необхідними для того, щоб реалізувати поставлену ціль (система такого виду отримала назву «експертна система»). Логічний підхід також використовує нечітку логіку для досягнення кращої виразності. Головна відмінність цього напряму полягає в тому, що істинність вислову може приймати не лише значення «так» або «ні» (1 або 0), а ще й проміжні значення — «не знаю» (0,5), «швидше так, ніж ні» (0,75), «швидше ні, ніж так» (0,25). Такий підхід є більш схожим на те, як мислить людина, оскільки вона зрідка відповідає лише «так» або «ні».
2. Структурний підхід. Під ним зазвичай розуміють спроби побудови ШІ шляхом моделювання структури людського мозку. Першою зі спроб такої побудови ШІ став перцептрон Френка Розенблата. Головною моделюючою структурною одиницею в перцептронах є

нейрон. Згодом з’явилися й інші моделі, які зараз називаються нейронні мережі (НМ) і їхні реалізації — нейрокомп'ютери. Ці моделі можуть бути різними за будовою окремих нейронів, топологією їх взаємозв'язків, а також алгоритмами навчання. У більш широкому розумінні цей підхід відомий як конекціонізм.

3. Еволюційний підхід. При побудові системи ШІ за даним методом основна увага зосереджується на побудові початкової моделі і правилах, за якими вона може змінюватися (еволюціонувати). Модель може бути створена за різними методами. Після цього комп'ютер перевіряє моделі і потім обирає кращі з них, саме за цими моделями будуть генеруватися нові. Одним з класичних прикладів еволюційних алгоритмів можна назвати генетичний алгоритм.
4. Імітаційний підхід. Даний підхід є досить типовим для кібернетики з одним із її базових понять - чорний ящик. Об'єкт, поведінка якого імітується, являє собою «чорний ящик». Не має значення, які моделі всередині об’єкту та як він взагалі діє, найважливішим є те, щоби наша розроблена модель в подібних ситуаціях мала таку саму поведінку. Таким чином тут моделюється здатність людини копіювати те, що роблять інші, без поділу на елементарні операції і формального опису дій.

Деякі з цих напрямків вчені іноді намагаються об’єднати в рамках гібридних інтелектуальних систем. Експертні правила висновків, можуть бути згенеровані НМ, а за допомогою статистичного вивчення можуть бути отримані побіжні правила.

Проаналізувавши розвиток ШІ, можна зазначити великий напрям, що має назву «моделювання міркувань». Саме за цим шляхом багато років розвивався ШІ, і на даний момент це одна з найрозвиненіших областей в сучасному ШІ. Моделювання міркувань має на увазі створення символьних систем, на вході

яких поставлена деяка задача, а на виході очікується її розв'язок. Як правило, запропонована задача уже переведена на математичну форму (формалізована), але вона або не має алгоритму розв'язання, або цей алгоритм занадто складний, трудомісткий тощо. В цей напрям входять: автоматизоване доведення теорем, прийняття рішень і теорія ігор, планування і диспетчеризація, прогнозування.

1.2 Штучний інтелект та логічне програмування

Логічне програмування- парадигма програмування, а також розділ дискретної математики, що вивчає методи і можливості цієї парадигми, засновані на виведенні нових фактів з даних фактів згідно із заданими логічними правилами. Логічне програмування засноване на теорії математичної логіки. Найвідомішою мовою логічного програмування є Prolog, що є за своєю суттю універсальною машиною виводу, що працює в припущенні замкненості системи фактів.

У широкому тлумаченні ЛП включає коло понять, методів, мов та систем, в основі якого лежить ідея опису задачі з сукупністю тверджень деякою логічною мовою та отримання розв'язання задачі шляхом побудови логічного висновку в деякій формальній дедуктивній системі. Класи формул, що використовуються для опису задач, методи виведення та моделі обчислень, що використовуються для опису задач, методи виведення та моделі обчислень, що використовуються в таких системах, дуже різноманітні, тому різні їх комбінації утворюють специфічні стилі програмування: концептуальна, функціональна (аплікативна) та ін.

Найбільші практичні результати досягнуті в системах програмування, де як логічні програми використовуються спеціальні класи логічних формул: хорнівські диз'юнкти, а як спосіб їх використання застосовуються спеціальні методи логічного виведення - варіанти методу резолюції.

1.3 Числення висловлень

Числення висловлень (логіка висловлень, пропозиційна логіка) — це розділ символьної логіки, що вивчає складні висловлення, що утворюються з більш простих, та їх взаємовідносини. На відміну від логіки предикатів, пропозиційна логіка не розглядає внутрішню структуру простих висловлювань, вона лише враховує, з допомогою яких зв’язків і в якому порядку прості висловлення зчленовуються у складні.

Численням висловлень є система $\mathcal{L} = \mathcal{L}(\mathrm{A},\ \Omega,\ \mathrm{Z},\ \mathrm{I})$, де:

- Множина A — це скінченна множина символів висловлень.
- Множина Ω — скінченна множина логічних зв'язок (логічних операторів). Дану множина може бути розбита на підмножини:

  $\Omega = \Omega_0 \cup \Omega_1 \cup \ldots \cup \Omega_j \cup \ldots \cup \Omega_m.$

  У цьому розбитті Ω_j є множина операторів арності j (також позначено $\Omega_0 = \{0, 1\}.$).

  Найчастіше використовуються оператори:

  $\Omega_1 = \{\neg\},$

  $\Omega_2 \subseteq \{\wedge, \vee, \rightarrow, \leftrightarrow\}.$
- Множина Z — скінченна множина правил виводу, за допомогою яких можна отримувати одні формули з інших.
- Множина I — скінченна множина, елементи якої називаються аксіомами. В окремих прикладах дана множина може бути пустою.

Виведення формул і теорем відбувається наступним чином:

Нехай Σ деяка множина формул $\mathcal{L}$, а A — деяка задана формула. Кажуть, що формула A виводиться з множини формул Σ (позначається як $\Sigma \vdash A$), якщо

існує така скінченна послідовність формул $A_1, A_2 \ldots A_n = A,$ де для кожної формули A_i:

- A_i є аксіомою, або
- A_i належить множині Σ або
- A_i виводиться з попередніх формул послідовності за допомогою котрогось із правил виводу.

Якщо при цьому множина Σ — пуста (формула А виводиться лише за допомогою аксіом і правил виводу), то формула А називається теоремою (для цього використовується позначення $\vdash A$)

1.3.1 Мова логіки висловлень

Мова логіки висловлювань (пропозиційна мова) - формалізована мова, за допомогою якої відбувається аналіз логічної структури складних висловлювань.

Мовою числення висловлень є множина формул, що визначаються рекурсивно за допомогою наступних правил:

1. кожен з елементі множини А є формулою;
2. якщо $p_1, p_2, \ldots, p_j$ є формулами та $f \in \Omega_j$, тоді $(f(p_1, p_2, \ldots, p_j))$ теж є формулою.
3. інших формул, окрім побудованих за першим та другим правилами немає.

Вихідні символи, або алфавіт мови логіки висловлювань, розділені на наступні три категорії:

- пропозиційні літери (пропозиційні змінні). Це змінні, якими замінюють елементарні логічні висловлення.

$p, q, r, s, t, p_1, q_1, r_1, s_1, t_1, \ldots$

- логічні знаки (логічні спілки) (таб 1.1):

Таблиця 1.1.

Логічні знаки в алфавіті мови логіки висловлень

Символ	**Значення**
$\neg$	Знак заперечення
$\wedge$ або &	Знак кон'юнкції («І»)
$\vee$	Знак диз'юнкції («АБО»)
$\oplus$ *або* $\dot{\vee}$	Знак суворої диз'юнкції («виключе АБО»)
$\rightarrow$	Знак імплікації
$\leftrightarrow$ або ~ або $\equiv$	Знак еквівалентності

- допоміжні символи: ліва дужка «(», права дужка «)», кома «,».

Роль структурних утворень, аналогічних елементарним і складним висловлюванням, грають у цій мові формули. Пропозиціональна формула - слово мови логіки висловлювань, тобто кінцева послідовність знаків алфавіту, побудована за викладеними нижче правилами і яка утворює закінчений вираз мови логіки висловлювань . Великі латинські літери A, B та інші, які вживаються у визначенні формули, належать не мові логіки висловлювань, а її метамові, тобто мові, яка використовується для опису самої мови логіки висловлювань. Вислови, що містять металітери $\neg A$, $(A \rightarrow B)$ та інші - не пропозиційні формули, а схеми формул. Наприклад, вираз $(A \wedge B)$ є схема, під яку підходять формули $(p \wedge q)$, $(p \wedge (r \vee s))$ та інші.

Індуктивне визначення формули логіки висловлювань:

1. пропозиційна змінна є формулою;

2. якщо A - довільна формула, то $\neg A$ - теж формула;
3. якщо A і B — довільні формули, то $(A \to B)$, $(A \wedge B)$, $(A \leftrightarrow B)$, $(A \vee B)$, $(A \dot{\vee} B)$ - теж формули.

Оскільки в побудованих за визначенням формулах виявляється занадто багато дужок, які не завжди є необхідними для однозначного розуміння формули, математики ухвалили угоди про дужки, за якими деякі з дужок можна опускати. Записи з опущеними дужками відновлюються за такими правилами.

- Якщо опущені зовнішні дужки, вони відновлюються.
- Якщо поруч стоять дві кон'юнкції або диз'юнкції (наприклад, $A \wedge B \wedge C$), то в дужки полягає спочатку сама ліва частина (тобто дві підформули з зв'язкою між ними).
- Якщо поруч стоять різні зв'язки, то дужки розставляються згідно з пріоритетами: $\neg$, $\wedge$, $\vee$, і $\to$ (від найвищого до нижчого).

Коли говорять про довжину формули, то мається на увазі довжина формули, що відновлюється, а не скороченого запису.

1.3.2 Семантика числення висловлень

Основним завданням логіки висловлювань є встановлення істинного значення формули, якщо дано істинні значення змінних, що входять до неї. Істинне значення формули в такому разі визначається індуктивно за допомогою таблиць істинності.

Нехай $\mathbb{B}$ - множина всіх істинних значень $\{0, 1\}$, а $Vars$ - множина пропозиційних змінних. Тоді інтерпретацію (або модель) мови логіки висловлювань можна подати у вигляді відображення $M: Vars \to \mathbb{B}$, яке кожну пропозиційну змінну p зіставляє з істинним значенням $M(p)$.

Найчастіше вживані оператори задаються за допомогою таблиць істинності:

p	$\neg p$
1	0
0	1

Рисунок 1.1 Таблиця істинності для операції «НЕ»

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

Рисунок 1.2 Таблиця істинності для операції «І»

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

Рисунок 1.3 Таблиця істинності для операції «АБО»

p	q	$p \to q$
1	1	1
1	0	0
0	1	1
0	0	1

Рисунок 1.4 Таблиця істинності для операції імплікації

Зважаючи на спосіб побудови формул, кожна формула при деякому заданню істинності отримує певне значення 0 або 1. Значення найпростіших формул для різних завдань істинності можна обчислювати за допомогою таблиць істинності. Наприклад:

p	q	r	$(p \vee q)$	$\neg(p \vee q)$	$(p \to r)$	$\neg(p \vee q) \to (p \to r)$
1	1	1	1	0	1	1
1	1	0	1	0	0	1
1	0	1	1	0	1	1
1	0	0	1	0	0	1
0	1	1	1	0	1	1
0	1	0	1	0	1	1
0	0	1	0	1	1	1
0	0	0	0	1	1	1

Рисунок 1.5 Приклад таблиці істинності для формул

Якщо для деякого задання істинності I формула A набуває значення 1, то кажуть, що формула A задовольняє задання I. Формула, що задовольняє усі можливі задання істинності (як формула $\neg(p \vee q) \to (p \to r)$) називається тавтологією. Якщо Σ — деяка множина формул то кажуть, що дана множина задовольняє задання істинності, якщо це задання задовольняє кожна формула цієї множини. Якщо для деякої формули A з того, що множина Σ задовольняє заданню істинності випливає що A задовольняє цьому заданню то формула A

називається логічним наслідком множини Σ (позначається $\Sigma \models A$). У випадку якщо множина Σ є пустою, формула є тавтологією.

1.3.3 Тотожні формули (тавтології) в численні висловлень

Формула є тотожно істинною, якщо вона істинна при будь-яких значеннях змінних, що входять до неї (тобто, при будь-якій інтерпретації). Далі перераховані кілька досить відомих прикладів тотожно істинних формул логіки висловлювань:

- **закони де Моргана**:

$\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$

$\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$

- **закон контрапозиції**:

$(p \to q) \leftrightarrow (\neg q \to \neg p)$

- **закони поглинання**:

$p \vee (p \wedge q) \leftrightarrow p$

$p \wedge (p \vee q) \leftrightarrow p$

- **закони дистрибутивності**:

$p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$

$p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$

1.3.4 Основні та похідні форми аргументів

Основні та похідні форми аргументів надані в таб. 1.2.

Таблиця 1.2

Форми аргументів

Назва	**Послідовність**	**Опис**
Modus Ponens	$((p \to q) \land p) \vdash q$	Якщо p то q; p; тому q
Modus Tollens	$((p \to q) \land \neg q) \vdash \neg p$	Якщо p то q; не q; тому не p
Hypothetical Syllogism	$((p \to q) \land (q \to r)) \vdash (p \to r)$	Якщо p то q; якщо q то r; тому, якщо p то r
Disjunctive Syllogism	$((p \lor q) \land \neg p) \vdash q$	Або p або q, або обидва; не p; тому, q
Constructive Dilemma	$((p \to q) \land (r \to s) \land (p \lor r)) \vdash (q \lor s)$	Якщо p то q; і якщо r то s; але p або r; тому q або s
Destructive Dilemma	$((p \to q) \land (r \to s) \land (\neg q \lor \neg s)) \vdash (\neg p \lor \neg r)$	Якщо p то q; і якщо r то s; але не q або не s; тому не p або не r
Bidirectional Dilemma	$((p \to q) \land (r \to s) \land (p \lor \neg s)) \vdash (q \lor \neg r)$	Якщо p то q; і якщо r то s; але p або не s; тому q або не r
Simplification	$(p \land q) \vdash p$	p і q правда; тому p правда
Conjunction	$p, q \vdash (p \land q)$	p і q правда окремо; тому вони правда в кон'юнкції
Addition	$p \vdash (p \lor q)$	p правда; тому диз'юнкція (p або q) правда
Composition	$((p \to q) \land (p \to r)) \vdash (p \to (q \land r))$	Якщо p то q; і якщо p то r; тому якщо p правда то q і r правда
De Morgan's Theorem (1)	$\neg(p \land q) \vdash (\neg p \lor \neg q)$	Заперечення (p і q) дорівнює (не p або не q)
De Morgan's Theorem (2)	$\neg(p \lor q) \vdash (\neg p \land \neg q)$	Заперечення (p або q) дорівнює (не p і не q)
Commutation (1)	$(p \lor q) \vdash (q \lor p)$	(p або q) дорівнює (q або p)
Commutation (2)	$(p \land q) \vdash (q \land p)$	(p і q) дорівнює (q і p)

Продовження таблиці 1.2

Commutation (3)	$(p \leftrightarrow q) \vdash (q \leftrightarrow p)$	(*p* дорівнює *q* дорівнює (*q* дорівнює *p*)
Association (1)	$(p \vee (q \vee r)) \vdash ((p \vee q) \vee r)$	*p* або (*q* або *r*) дорівнює (*p* або *q*) або *r*
Association (2)	$(p \wedge (q \wedge r)) \vdash ((p \wedge q) \wedge r)$	*p* і (*q* і *r*) дорівнює (*p* і *q*) і *r*
Distribution (1)	$(p \wedge (q \vee r)) \vdash ((p \wedge q) \vee (p \wedge r))$	*p* і (*q* або *r*) дорівнює (*p* і *q*) або (*p* і *r*)
Distribution (2)	$(p \wedge (q \vee r)) \vdash ((p \wedge q) \vee (p \wedge r))$	*p* або (*q* і *r*) дорівнює (*p* або *q*) і (*p* або *r*)
Double Negation	$p \vdash \neg\neg p$	*p* дорівнює запереченню не *p*
Transposition	$(p \rightarrow q) \vdash (\neg q \rightarrow \neg p)$	Якщо *p* то *q* дорівнює якщо не *q* то не *p*
Material Implication	$(p \rightarrow q) \vdash (\neg p \vee q)$	Якщо *p* то *q* дорівнює не *p* або *q*
Material Equivalence (1)	$(p \leftrightarrow q) \vdash ((p \rightarrow q) \wedge (q \rightarrow p))$	(*p* якщо і тільки якщо *q*) дорівнює (якщо *p* правда то *q* правда) і (якщо *q* правда то *p* правда)
Material Equivalence (2)	$(p \leftrightarrow q) \vdash ((p \wedge q) \vee (\neg p \wedge \neg q))$	(*p* якщо і тільки якщо *q*) дорівнює або (*p* і *q* правда) або (і *p* і *q* брехня)
Material Equivalence (3)	$(p \leftrightarrow q) \vdash ((p \vee \neg q) \wedge (\neg p \vee q))$	(*p* якщо і тільки якщо *q*) дорівнює., і (*p* або не *q* правда) і (не *p* або *q* правда

Продовження таблиці 1.2

Exportation	$((p \wedge q) \rightarrow r) \vdash (p \rightarrow (q \rightarrow r))$	з (якщо *p* ф *q* правда то *r* правда) можна довести (якщо *q* правда то *r* правда, якщо *p* правда)
Importation	$(p \rightarrow (q \rightarrow r)) \vdash ((p \wedge q) \rightarrow r)$	Якщо *p* то (якщо *q* то *r*) дорівнює якщо *p* і *q* то *r*

Tautology (1)	$p \vdash (p \lor p)$	p правда дорівнює p правда або p правда
Tautology (2)	$p \vdash (p \land p)$	p правда дорівнює p правда і p правда
Tertium non datur (Law of Excluded Middle)	$\vdash (p \lor \neg p)$	p або не p правда
Law of Non-Contradiction	$\vdash \neg(p \land \neg p)$	p і не p брехня, правдиве твердження

1.3.5 Основні проблеми числення висловлень

Для обґрунтування будь-якої аксіоматичної теорії необхідно розглянути наступні 4 проблеми:

1. Проблема несуперечливості.

   Визначення: Нехай дано деяку формальну аксіоматичну теорію. Кажуть, що побудована модель цієї теорії, якщо всім символам алфавіту приписується певний зміст, що описує певну неформальну теорію та співвідношення між елементами цієї теорії.

   Формальна аксіоматична теорія називається категоричною, якщо будь-які її 2 моделі ізоморфні між собою, тобто між ними може бути встановлена взаємна відповідність.

   Формальна аксіоматична теорія називається послідовною щодо її моделі, якщо будь-яка теорема у формальній теорії є істинним твердженням для моделі.

   Формальна аксіоматична теорія числення висловлень називається внутрішньо несуперечливою, неможливо довести якусь теорему (формулу) разом із її запереченням.

   Формальна аксіоматична теорія називається синтаксично несуперечливою , якщо вона містить принаймні деяку формулу, яка не є теоремою.

Теорема: Формальна аксіоматична теорія числення висловлень узгоджується з її моделлю пропозиційної алгебри.

Наслідок:

- Числення висловлень – внутрішньо-несуперечлива теорія.
- Числення висловлень – синтаксично-несуперечлива теорія.

2. Проблема повноти

Формальна аксіоматична теорія числення висловлень називається повною у вузькому сенсі, якщо додавання до системи аксіом цієї теорії хоча б однієї формули, яка не є теоремою, призводить до того, що теорія стає внутрішньо суперечливою.

Формальна аксіоматична теорія числення висловлень є повною у широкому розумінні або повною відносно своєї моделі, якщо будь-яка формула, істинна в моделі, є теоремою в цій теорії, або якщо можна довести будь-яку формулу, що тотожно вірна.

Теорема: Формальна аксіоматична теорія числення висловлень є повною відносно своєї моделі алгебри висловлень.

Теорема: Числення висловлень – це формальна аксіоматична теорія, повна у вузькому розумінні.

3. Проблема незалежності

Визначення: Нехай дана формальна аксіоматична теорія, кажуть, що деяка аксіома цієї теорії є незалежною, якщо її не можна довести методами самої теорії, як теорему.

Система аксіом формальної аксіоматичної теорії називається незалежною системою аксіом, якщо всі аксіоми незалежні.

Теорема: Система аксіом числення висловлень є незалежною.

Доведення: Для доведення незалежності деякої аксіоми числення висловлень використовують наступний підхід: будується модель формальної аксіоматичної теорії, в якій всі аксіоми, крім цієї, справджуються. Якщо доведено, що така модель ізоморфна стандартній моделі формальної аксіоматичної теорії, то робиться висновок, що аксіома не є незалежною; якщо ж такого ізоморфізму немає – незалежна.

4. Проблема розв'язності

Полягає в доведені існування алгоритму, який для будь-якої формули числення висловлень визначає чи можна її довести чи ні.

Теорема: Проблема розв`язності числення висловлень є розв`язною.

Теорема 1: Будь-яка тотожно істинна формула пропозиційної алгебри є теоремою числення висловлень.

Доведення: Нехай A - довільна формула числення числення висловлень. Побудуємо для неї таблицю істинності і розглянемо її останній стовпець. Якщо він містить лише одиниці, то A - тотожно істинна формула і, згідно з теоремою 1, є теоремою числення висловлень. Інакше (останній стовпчик таблиці істинності містить хоча б один нуль), A не є тавтологією, а отже, A не є теоремою.

1.4 Числення предикатів

Логіка першого порядку (числення предикатів) — це формальна система математичної логіки, в якій допускаються висловлення щодо змінних, фіксованих функцій, і предикатів. Це розширення пропозиційної логіки. У свою чергу, це окремий випадок логіки вищого порядку.

Мови логіки першого порядку будуються на основі сигнатури, що складається із набору функціональних символів $\mathcal{F}$ і набору символів-предикатів $\mathcal{P}$. З кожним функціональним і предикатним символом пов'язана арність (число агрументів). Крім того використовуються додаткові символи:

- Символи змінних $x, y, z, x_1, y_1, z_1, x_2, y_2, z_2,$ і т. д.,
- Пропозиційні зв'язки: $\vee, \wedge, \neg, \to$
- Квантори: загальності $\forall$ та існування $\exists$,
- Службові символи: дужки і кома.

Перелічені символи разом із символами з $\mathcal{P}$ і $\mathcal{F}$ утворюють алфавіт логіки першого порядку. Складніші конструкції визначаються індуктивно:

- Терм — це символ змінної, або має вид $f(t_1, \ldots, t_n)$, де f — функціональний символ арності n, а $t_1, \ldots, t_n$ — терми.
- Атом — має вид $p(t_1, \ldots, t_n)$, де p — предикатний символ арності n, а $t_1, \ldots, t_n$ — атоми.
- Формула — це або атом, або одна з наступних конструкцій: $\neg F, (F_1 \vee F_2), (F_1 \wedge F_2), (F_1 \to F_2), \forall x F, \exists x F$ — формули, а x — змінна.

Змінна x називається зв'язаною в формулі F, якщо F має вид $\forall x G$ або $\exists x G$, або може бути представлена в одній з форм $\neg H, (F_1 \vee F_2), (F_1 \wedge F_2), (F_1 \to F_2)$, причому x вже зв'язана в H, F_1 і F_2.

Якщо x не зв'язана в F, її називають незв'язаною в F. Формулу без незв'язаних змінних називають замкнутою формулою. Теорією першого порядку називають довільну множину замкнутих формул.

1.4.1 Аксіоматика числення предикатів

Наступна система логічних аксіом логіки першого порядку містить усі аксіоми числення висловлень та дві додаткові аксіоми:

1. $(A \to (B \to A))$
2. $((A \to (B \to C)) \to ((A \to B) \to (A \to C)))$
3. $((\neg A \to \neg B) \to (B \to A))$
4. $\forall x A \to A[t/x]$
5. $\forall x(A \to B) \to (A \to \forall x B)$, x не присутній в A в незвязаному стані

У четвертій аксіомі A[t/x] — формула, одержана внаслідок підстановки терма t замість змінної x в формулі A. Проте не в усіх випадках можлива підстановка деякого терма замість змінної. Умови існування такої заміни та її результат можна визначити індуктивно.

- Якщо A — атомарна формула, то терм t може замінити довільну змінну x цієї формули. Результат позначається A[t/x].
- Якщо A має вигляд $\neg B$ тоді підстановка t замість x можлива лише тоді, коли така підстановка можлива для формули B і A[t/x] тоді дорівнює $\neg B[t/x]$.
- Якщо A має вигляд $B \to C$, тоді підстановка t замість x можлива лише тоді, коли така підстановка можлива для формул B і C. A[t/x] тоді рівна $B[t/x] \to C[t/x]$.
- Якщо A має вигляд $\forall y B$, тоді підстановка t замість x можлива у випадках:
  - Змінна x зустрічається у формулі B лише у зв'язаному стані.
  - змінна y не зустрічається у термі t і підстановка t замість x можлива у формулі B. Тоді результат визначається наступним чином:
    - Якщо x дорівнює y, то A[t/x] дорівнює $\forall y B$

- Якщо x не дорівнює y, то [t/x] дорівнює $\forall y B[t/x]$

Окрім того є два правила виводу:

- Modus ponens:

$$\frac{A, A \to B}{B}$$

- Правило узагальнення (GEN):

$$\frac{A}{\forall x A}$$

Ці аксіоми і правила виводу є схемами і A, B, C можна заміняти довільними формулами.

В цій аксіоматиці використовуються лише дві пропозиційні зв'язки: $\neg$, $\to$ і квантор загальності $\forall$. Інші пропозиційні зв'язки і квантор існування можна визначити наступним чином:

- $(A \vee B)$ позначає $(\neg(A \to (\neg B)))$
- $(A \wedge B)$ позначає $((\neg A) \to B)$
- $(\exists x A)$ позначає $(\neg \forall x (\neg A))$

Всі перераховані вище аксіоми називаються логічними. Якщо інших аксіом немає, то така формальна система називається обчисленням предикатів першого порядку. Числення предикатів першого порядку є прикладом теорії першого порядку. Усі теорії першого порядку визначаються подібно до числення предикатів першого порядку, однак вони мають додаткові аксіоми, які також називають власними аксіомами теорії.

1.4.1 Виведення формул і теорем

Нехай Σ деяка множина формул мови першого порядку, а F — деяка задана формула. Тоді кажуть, що формула F є похідною з множини формул Σ (позначається $\Sigma \vdash A$), якщо існує така скінченна послідовність формул $A_1, A_2 \ldots A_n = A$, де для кожної формули A_i :

1. A_i є аксіомою, або
2. A_i належить до множини Σ або
3. A_i отримується з попередніх формул послідовності за допомогою одного з правил виводу.

Якщо при цьому множина Σ — порожня (формула A виводиться лише за допомогою аксіом і правил виводу), то формула A називається теоремою (для цього використовується позначення $\vdash A$).

Множина Σ формул називається несуперечливою, якщо для довільної формули A не виконується одночасно $\Sigma \vdash A$ і $\Sigma \vdash \neg A$.

1.4.2 Властивості числення предикатів

1. Правильність і повнота. Наведена вище система аксіом і правил виводу є правильною, тобто для будь-якого набору формул Σ із $\Sigma \vdash A$ випливає $\Sigma \models A$. Також ця система є повною: із $\Sigma \models A$ випливає $\Sigma \vdash A$. Зокрема, з цих тверджень випливає, що для числення предикатів першого порядку загальнозначимі формули збігаються із теоремами формальної системи.

Крім того, в будь-якій теорії першого порядку всі формули, виведені в ній, збігаються з формулами, істинними в усіх моделях цієї теорії.

2. Компактність. Деякий набір формул виконується тоді і тільки тоді, коли виконуються всі його скінченні підмножини.
3. Нерозв'язність. На відміну від пропозиційної логіки, логіка першого порядку є нерозв'язною, якщо існує хоча б один предикат арності принаймні 2 (за винятком рівності), тобто немає ефективного методу визначення «чи не існує виведення деякої формули?» у певній теорії першого порядку.

1.5 Методи та алгоритми перевірки виводимості формул

Нехай задана сукупність ППФ $\mathrm{A} = \{A_1, A_2, ..., A_n\}$, яка називається гіпотезами, та ППФ – В. «И» називається логічним наслідком $\mathrm{A} = \{A_1, A_2, ..., A_n\}$ (записується як (А⊢В)), якщо $F = (A_1 \,\&\, A_2, ..., \,\&\, A_n) \rightarrow B$ є тавтологією (тотожно істинним висловом).

Таким чином, завдання перевірки виведення зводиться до перевірки $\mathrm{A} \rightarrow B$ на тотожну істинність. Існує кілька десятків методів та алгоритмів встановлення тотожної істинності логічної формули.

1. Алгоритм істинних таблиць (АІТ) для F = A ® B.

   АІТ зводиться до послідовної підстановки всіх можливих інтерпретацій (наборів «істина» та «брехня») змінних, що входять до F. Алгоритм зупиняється, якщо значення F = «брехня» (F не здійсненна, а значить B не виводиться з A на усіх інтерпретаціях); істина (F здійсненна на усіх інтерпретаціях, отже F суть тавтологія і A⊢B . Такий алгоритм вимагає

у найгіршому разі 2^n підстановок (2^n можливих інтерпретацій), де «n» число змінних, які входять у формулу F.

2. Метод резолюцій

   Алгоритми докази виведення A⊢B, побудовані на основі цього методу, застосовуються в багатьох системах штучного інтелекту, а також є фундаментом, на якому побудовано мову логічного програмування «Пролог».

   Правило резолюцій — це правило виводу, що сходить до методу доказу теорем через пошук протиріч; використовується в логіці висловлювань і логіці предикатів першого порядку. Правило резолюцій, що застосовується послідовно для списку резольвент, дозволяє відповісти на питання, чи існує у вхідній множині логічих виразів протиріччя. Правило резолюцій запропоновано в 1930 році в докторській дисертації Жака Ербрана для доведення теорем у формальних системах першого порядку. Правило розроблено Джоном Аланом Робінсоном в 1965 році.

3. Алгоритм Куайна

   Метод Куайна — спосіб мінімізації функцій алгебри логіки. Представляє функції у вигляді ДНФ або КНФ з мінімальною кількістю членів і з мінімальним набором змінних.

   Ідея: при послідовних підстановках значень змінних можна зменшити довжину формули, виходячи із сукупності проведених перевірок істинності F, тим самим скорочувати кількість змінних для перевірки.

   Вводиться поняття дерева випробувань, яке насправді є граф всіх інтерпретацій формули. Куайн назвав його "семантичним деревом".

1.6 Метод перевірки виводимості формул Куайна – Мак-Класкі

1.6.1 Загальні положення методу Куайна

Метод Куайна має чітко сформульований алгоритм виконання окремих операцій і, отже, може бути використаний для реалізації на ЕОМ. Мінімізація цим методом вимагає багато часу через необхідність попарного порівняння членів логічної функції один з одним. У разі мінімізації за методом Куайна передбачається, що початкова функція задана в ДДНФ або ДКНФ.

Сам метод проходить в два етапи:

- Перехід від канонічної форми (ДДНФ або ДКНФ) до скороченої форми.
- Перехід від скороченої до мінімальної форми.

Перший етап (отримання скороченої форми). Уявімо, що задана функція f представлена в СДНФ. Для реалізації першого етапу перетворення виконуються дві дії:

- Операція склеювання;
- Операція поглинання.

Операція склеювання зводиться до пошуку пар членів, що відповідають формі $w \cdot x$ або $w \cdot \overline{x}$, та перетворення їх у наступні вирази $w \cdot x \vee w \cdot \overline{x} = w \cdot (x \vee \overline{x}) = w$. Результати склеювання w тепер грають роль додаткових членів. Необхідно знайти всі можливі пари членів (кожен член із кожним).

Потім виконується операція поглинання. Вона заснована на рівності $w \vee w \cdot z = w \cdot (1 \vee z) = w$ (член w поглинає вираз $w \cdot z$). В результаті цієї дії всі члени, поглинені іншими змінними, результати яких були отримані в операції склеювання, видаляються з логічного виразу.

Обидві операції першого етапу можуть виконуватися до тих пір, поки це може бути здійснено.

Застосування цих операцій показано в таблиці 1.3:

Таблиця 1.3

Застосування операцій склеювання та поглинання

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

СДНФ має наступний вигляд:

$$f(x_1, x_2, x_3) = \overline{x_1} \cdot x_2 \cdot \overline{x_3} \vee x_1 \cdot \overline{x_2} \cdot \overline{x_3} \vee x_1 \cdot \overline{x_2} \cdot x_3 \vee x_1 \cdot x_2 \cdot \overline{x_3} \vee x_1 \cdot x_2 \cdot x_3$$

Результат операції склеювання необхідний для перетворення функції на другому етапі (поглинання)

$$f(x_1, x_2, x_3) = \cancel{\overline{x_1} \cdot x_2 \cdot \overline{x_3}} \vee \cancel{x_1 \cdot \overline{x_2} \cdot \overline{x_3}} \vee \cancel{x_1 \cdot \overline{x_2} \cdot x_3} \vee \cancel{x_1 \cdot x_2 \cdot \overline{x_3}} \vee \cancel{x_1 \cdot x_2 \cdot x_3}$$

$$= x_2 \cdot \overline{x_3} \vee x_1 \cdot \overline{x_2} \vee x_1 \cdot \overline{x_3} \vee x_1 \cdot x_3 \vee x_1 \cdot x_2$$

Членами результату склеювання є $x_2 \cdot \overline{x_3} \vee x_1 \cdot \overline{x_2} \vee x_1 \cdot \overline{x_3} \vee x_1 \cdot x_3 \vee x_1 \cdot x_2$

Член $x_2 \cdot \overline{x_3}$ поглинає ті члени вихідного виразу, які містять $x_2 \cdot \overline{x_3}$, тобто перший і четвертий. Ці члени видаляються. Член $x_1 \cdot \overline{x_2}$ поглинає другий та третій, а член $x_1 \cdot x_3$ – п'ятий член вихідного виразу.

В результаті повторення обох операцій отримуємо наступне:

$$f(x_1, x_2, x_3) = x_2 \cdot \overline{x_3} \vee \cancel{x_1 \cdot \overline{x_2}} \vee \cancel{x_1 \cdot \overline{x_3}} \vee \cancel{x_1 \cdot x_3} \vee \cancel{x_1 \cdot x_2} \vee x_1$$

Тут склеюється пара членів $x_1 \cdot \overline{x_2}$ і $x_1 \cdot x_2$ (склеювання пари членів $x_1 \cdot \overline{x_3}$ і $x_1 \cdot x_3$ призводить до того ж результату), результат склеювання x_1 поглинає 2-, 3-, 4-, 5-й члени вираження. Подальші операції склеювання та поглинання виявляються неможливими, зменшена форма виразу заданої функції (в даному випадку вона збігається з мінімальною формою) $f(x_1, x_2, x_3) = x_2 \cdot \overline{x_3} \vee x_1$

Члени скороченої форми (у нашому випадку це $x_2 \cdot \overline{x_3}$ і x_1) називаються простими імплікантами функції. У результаті, отримуємо найпростіший вираз, якщо порівнювати його з початковою версією - ДДНФ. Структурна схема такого елемента показана малюнку нижче.

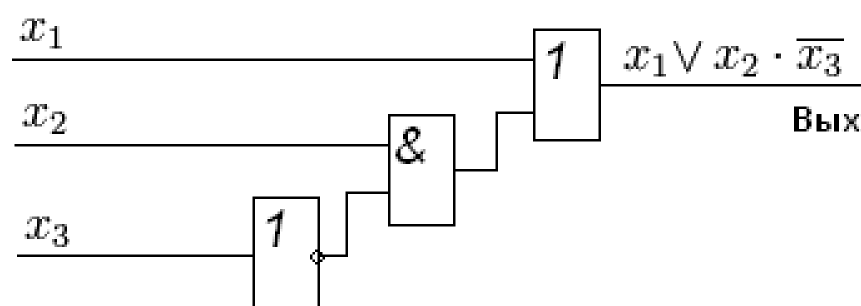


Рисунок 1.6. Структурна схема функції.

Другий етап (табличний) (отримання мінімальної форми). Як на першому етапі, результуюча рівність може містити члени, усунення яких жодним чином не вплине на кінцевий результат. Наступним етапом мінімізації є видалення

таких змінних. Таблиця нижче (таблиця 1.4) містить значення істинності функції. По ній буде зібрано наступну ДДНФ.

Таблиця 1.4

СДНФ для другого етапу метода Куайна

x_1	x_2	x_3	x_4	$f(x_1, x_2, x_3, x_4)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

СДНФ, зібрана за цією таблицею виглядає так:

$$f(x_1, x_2, x_3, x_4) = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4} \vee \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4 \vee \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} \vee$$

$$\overline{x_1} \cdot x_2 \cdot x_3 \cdot \overline{x_4} \vee x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4} \vee x_1 \cdot x_2 \cdot x_3 \cdot x_4$$

$$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \vee \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_4} \vee \overline{x_1} \cdot x_3 \cdot \overline{x_4} \vee x_2 \cdot x_3 \cdot \overline{x_4} \vee x_1 \cdot x_2 \cdot x_3$$

Члени цього виразу є простими імплікантами виразу. Перехід від скороченої форми до мінімальної здійснюється за допомогою імплікантної матриці.

Члени ДДНФ заданої функції вписуються в стовпці, а прості імпліканти, тобто члени зменшеної форми, вписуються в рядки. Зазначаються стовпці

членів ДДНФ, які поглинаються окремими простими імплікантами. У наступній таблиці (таблиця 1.5) проста імпліканта $\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$ поглинає члени $\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$ та $\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4$ (у першому та у другому стовпцях поставлені хрестики).

Таблиця 1.5

Поглинання членів формули імплікантою

Проста імпліканта	$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$	$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4$	$\overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4}$	$\overline{x_1} \cdot x_2 \cdot x_3 \cdot \overline{x_4}$	$x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4}$	$x_1 \cdot x_2 \cdot x_3 \cdot x_4$
$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$	×	×				
$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_4}$	×		×			
$\overline{x_1} \cdot x_3 \cdot \overline{x_4}$			×	×		
$x_2 \cdot x_3 \cdot \overline{x_4}$				×	×	
$x_1 \cdot x_2 \cdot x_3$					×	×

Друга імпліканта поглинає перший і третій члени ДДНФ (зазначено хрестиками) і т. д. Імпліканти, які не можна виключити, утворюють ядро. Такі імпліканти визначаються наведеною вище матрицею. Для кожного з них є принаймні один стовпець, який перекривається цим імплікантом.

У прикладі імпліканти $\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$ і $x_1 \cdot x_2 \cdot x_3$ (ними перекриваються другий і шостий стовпці) утворюють ядро. Виключення із скороченої форми одночасно всіх імплікантів, що не входять в ядро, неможливе, тому що виключення однієї з імплікантів може перетворити іншу на вже незайвий член.

Щоб отримати мінімальну форму, достатньо вибрати з-поміж імплікантів, що не входять до ядра, таку мінімальну їх кількість з мінімальною кількістю літер у кожній із цих імплікантів, яка забезпечить перекриття всіх стовпців, які не перекриваються члени ядра. У прикладі необхідно покрити третій і четвертий

стовпці матриці імплікантами, які не входять до ядра. Цього можна досягти різними способами, але оскільки необхідно вибрати мінімальну кількість імплікантів, то, очевидно, для перекриття цих стовпців слід вибрати імпліканту $\overline{x_1} \cdot x_3 \cdot \overline{x_4}$.

Мінімальна диз'юнктивна нормальна форма (МДНФ) заданої функції:

$$f(x_1, x_2, x_3, x_4) = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \vee x_1 \cdot x_2 \cdot x_3 \vee \overline{x_1} \cdot x_3 \cdot \overline{x_4} \quad (1.1)$$

Структурна схема, що відповідає цьому виразу, наведена на малюнку нижче (рис. 1.7).

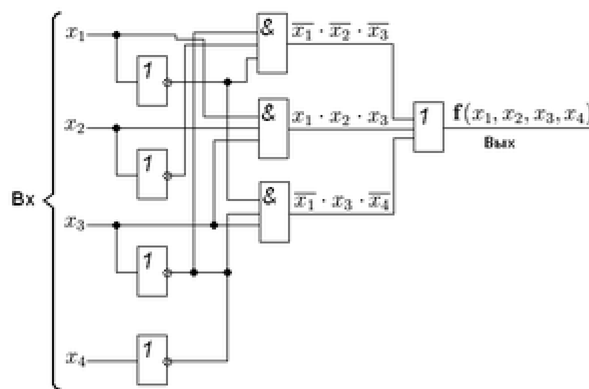


Рисунок 1.7. Структурна схема, що відповідає виразу в МДНФ при мінімізації функції методом Куайна.

Перехід від скороченої схеми до МДНФ здійснювався шляхом усунення непотрібних членів – імплікант $\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_4}$ і $x_2 \cdot x_3 \cdot \overline{x_4}$. Покажемо допустимість подібного виключення членів із логічного виразу.

Імпліканти $\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_4}$ і $x_2 \cdot x_3 \cdot \overline{x_4}$ стають рівними log 1 відповідно при наступних наборах значень аргументів: $x_1 = 0$, $x_2 = 0$, $x_4 = 0$ і $x_2 = 1$, $x_3 = 1$, $x_4 = 0$.

Роль цих імплікант у виразі скороченої форми функції полягає лише в тому, щоб надати функції $f(x_1, x_2, x_3, x_4)$ значення 1 на заданих наборах значень аргументів . Однак при цих наборах функція дорівнює 1 з-за інших імплікант виразу. Дійсно, підставляючи набір значень, зазначених вище у формулу 1.1, отримуємо:

- при $x_1 = 0$, $x_2 = 0$, $x_4 = 0$
  $f(0, 0, x_3, 0) = 1 \cdot 1 \cdot \overline{x_3} \vee 0 \cdot 0 \cdot x_3 \vee 1 \cdot x_3 \cdot 1 = \overline{x_3} \vee x_3 = 1$;
- при $x_2 = 1$, $x_3 = 1$, $x_4 = 0$
  $f(x_1, 1, 1, 0) = \overline{x_1} \cdot 0 \cdot 0 \vee x_1 \cdot 1 \cdot 1 \vee \overline{x_1} \cdot 1 \cdot 1 = x_1 \vee \overline{x_1} = 1$;

1.6.2 Метод Куайна - Мак-Класкі. Загальні положення

Метод Куайна-Мак-Класкі - табличний метод мінімізації булевих функцій, запропонований Уіллардом Куайном і вдосконалений Едвардом Мак-Класкі. Являє собою спробу позбутися недоліків методу Куайна.

Алгоритм Куайна–МакКласкі функціонально ідентичний відображенню Карно, але таблична форма робить його більш ефективним для використання в комп’ютерних алгоритмах, а також дає детермінований спосіб перевірити, чи досягнута мінімальна форма булевої функції. Його іноді називають методом таблиць.

Хоча алгоритм Куайна-МакКласкі є більш практичним, ніж відображення Карно, коли має справу з більш ніж чотирма змінними, він також має обмежений діапазон використання, оскільки проблема, яку він вирішує, є NP-повною. Час роботи алгоритму Куайна–МакКласкі зростає в геометричній прогресії зі збільшенням кількості змінних. Для функції з n змінних кількість простих імплікант може досягати 3^n/ ln(n), наприклад для 32 змінних може бути понад 534 × 1012 простих імплікант. Функції з великою кількістю змінних необхідно мінімізувати за допомогою потенційно неоптимальних евристик.

Другий крок алгоритму зводиться до розв'язування задачі покриття множини; На цьому етапі алгоритму можуть виникнути NP-складні екземпляри цієї проблеми.

Алгоритм даного методу наступний:

1. Терми (кон'юнктивні у разі ДДНФ та диз'юнктивні у разі ДКНФ), на яких визначено функцію алгебри логіки (ФАЛ) записуються як їх бінарні еквіваленти;
2. Ці еквіваленти поділяються на групи, до кожної групи входять еквіваленти з рівною кількістю одиниць (нулів);
3. Проводиться попарне порівняння еквівалентів (термів) у сусідніх групах з метою формування термів нижчих рангів;
4. Складається таблиця, заголовки рядків у якій — початкові терміни, а заголовки стовпців — терміни низьких рангів;
5. Розміщуються мітки, що відображають поглинання термів вищих рангів (початкових доданків), а потім виконується мінімізація за методом Куайна.

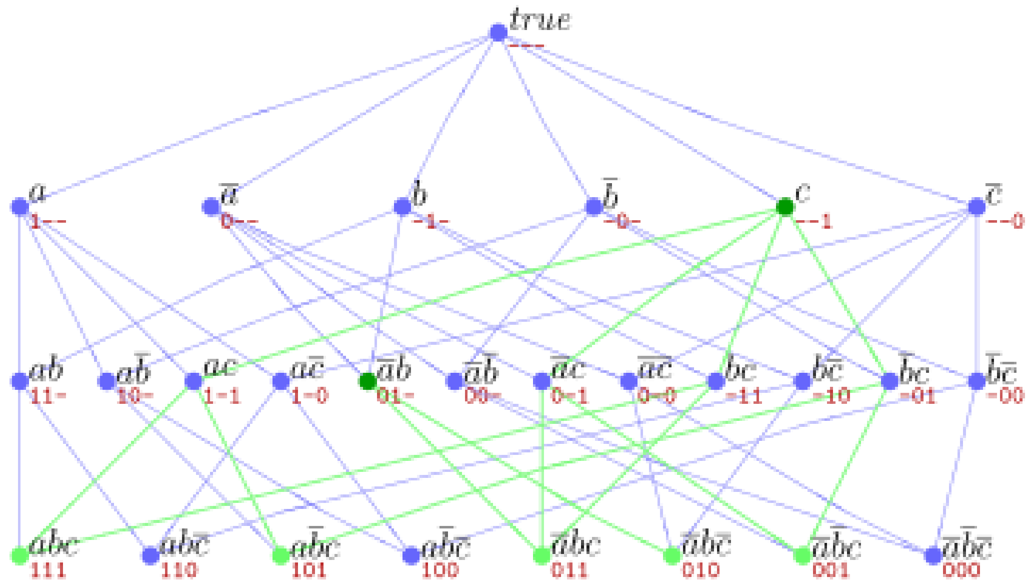


Рисунок 1.8. Діаграма Хассе графа пошуку алгоритму для 3 змінних. Враховуючи, напр. $S = \{abc, a\bar{b}c, \bar{a}bc, \bar{a}b\bar{c}, \bar{a}\bar{b}c\}$підмножина вузлів нижнього рівня (світло-зелений), алгоритм обчислює мінімальний набір вузлів (тут: темно-зелений), який охоплює саме S.

Специфіка методу Куайна - Мак-Класкі в порівнянні з методом Куайна в скороченні кількості попарних порівнянь щодо їх склеювання. Скорочення досягається рахунок вихідного розбиття термів на групи з рівною кількістю одиниць (нулів). Це дозволяє виключити порівняння, що свідомо не дають склеювання.

Висновки до першого розділу

На даному етапі розвитку інформаційних технологій штучний інтелект став одним з передових напрямків розвитку. Існують різні методи створення систем штучного інтелекту. Серед них виділяють 4 основні:

1. Логічний підхід. Основою для вивчення логічного підходу слугує алгебра логіки. Свого подальшого розвитку алгебра логіки отримала у

вигляді числення предикатів — в якому вона розширена за рахунок введення предметних символів, відношень між ними.

2. Структурний підхід. Спроби побудови ШІ шляхом моделювання структури людського мозку. Базується на перцептронах. Головною моделюючою структурною одиницею в перцептронах є нейрон.
3. Еволюційний підхід. Під час побудови системи ШІ за даним методом основну увагу зосереджують на побудові початкової моделі і правилах, за якими вона може змінюватися (еволюціонувати).
4. Імітаційний підхід. Цей підхід є класичним для кібернетики з одним із її базових понять чорний ящик. Об'єкт, поведінка якого імітується, якраз і являє собою «чорний ящик». Для нас не важливо, які моделі у нього всередині і як він діє, головне, щоби наша модель в аналогічних ситуаціях поводила себе без змін.

В даній роботі увага буде зосереджена на логічному підході, зокрема на численні висловлень, його особливостях, а також методах, за якими можна перевірити виводимість інформації.

Числення висловлень є формальною системою, в якій формули, що відповідають висловленням, можуть утворюватися шляхом з'єднання простих висловлень із допомогою логічних операцій, та система правил виводу, які дозволяють визначати певні формули як «теореми» формальної системи.

Серед методів та алгоритмів перевірки виводимості формул можна виділити наступні:

- Алгоритм істинних таблиць (АІТ);
- Метод резолюцій;
- Алгоритм Куайна, який був згодом покращений Едвардом Мак-Класкі.

Для подальших досліджень взято до розгляду алгоритм Куайна – Мак-Класкі. Він є формалізованим на етапі знаходження простих імплікант методом Куайна. Основним виявленим недоліком при програмній реалізації даного методу є швидкість його роботи. Проте, були виявлені також і інші характерні недоліки, які будуть розглянуті в наступному розділі.

РОЗДІЛ 2.
Оптимізація роботи методу Куайна - Мак-Класкі

2.1 Проблеми пов'язані з реалізацією методу в середовищі .NET та можливі варіанти їх вирішення

В програмній реалізації даного методу було виявлено наступні недоліки його роботи:

- швидкість роботи алгоритму. Як нам вже відомо з попереднього розділу, швидкість роботи даного методу і справді є його найбільшим недоліком, тому отримання цього висновку є очікуваним.
- при великій кількості вхідних змінних в термах робота програми завершується через нестачу пам'яті.

Оскільки швидкість роботи алгоритму є його «вродженою» вадою, основна увага в даній роботі буде зосереджена на вирішенні проблем, пов'язаних з нестачею пам'яті при роботі алгоритму.

Слід зазначити, що проблема нестачі пам'яті виникає тільки в тому випадку, якщо оптимізація відбувається на наборі вхідних терм розміром трохи менше максимальної кількості 2^N, де N – кількість змінних. Якщо ж використовується скорочений (неповний) набір з кількістю $Q < Q_{пор} < 2^N$, то проблема не виникає доти, поки розмір вхідного набору даних не переросте деякий поріг $Q_{пор}$, критичний для алгоритму.

В якості вирішення виявлених проблем пропонуються наступні рішення:

1. Визначення вузлів троїчного дерева як структур.
2. Додавання до програми пулу об'єктів.
3. Явний виклик методів Garbage Collector.

Розглянемо кожен з варіантів більш детально та подивимось на можливі теоретичні результати їх застосування.

2.2.1 Визначення вузлів троїчного дерева як структур

В реалізації методу було використано троїчне дерево, а його вузли були представлені у вигляді класів мови C#. Рішення у вигляді заміни класів на структури може бути застосоване для вирішення проблеми з нестачею пам’яті при запуску реалізації алгоритму Куайна – Мак-Класкі.

Розглянемо основні відмінності класів від структур з точки зору розміщення в пам’яті та проаналізуємо можливі переваги застосування структур замість класів.

Структури синтаксично дуже схожі на класи, але існує важлива відмінність, яка полягає в тому, що клас є посилальним типом даних, а структури – значущим типом. Отже, класи завжди створюються у так званій купі, а структури створюються в стеку.

Стек та купа відносяться до різних сегментів оперативної пам'яті.

Стек – це область оперативної пам'яті, що створюється для кожного потоку. Він працює в порядку LIFO (Last In, First Out), тобто останній доданий у стек шматок пам'яті буде першим у черзі на виведення зі стека.

Щоразу, коли функція оголошує нову змінну, вона додається в стек, а коли ця змінна зникає з області видимості, вона автоматично видаляється зі стека. Коли стекова змінна звільняється, ця область пам'яті стає доступною для інших стекових змінних.

Через таку природу стека управління пам'яттю виявляється дуже логічним і простим для виконання на ЦП; це призводить до високої швидкості, особливо

оскільки час циклу оновлення байта стека дуже малий, тобто, цей байт найімовірніше прив'язаний до кешу процесора.

Тим не менш, у такої жорсткої форми управління є і недоліки. Розмір стека – це фіксована величина, і перевищення ліміту виділеної на стеку пам'яті призведе до переповнення стека. Розмір задається при створенні потоку, і кожна змінна має максимальний розмір, що залежить від типу даних.

Це дозволяє обмежувати розмір деяких змінних (наприклад, цілих), і змушує заздалегідь оголошувати розмір більш складних типів даних (наприклад, масивів), оскільки стек не дозволить їм змінити його. Крім того, змінні, розташовані в стеку, є локальними.

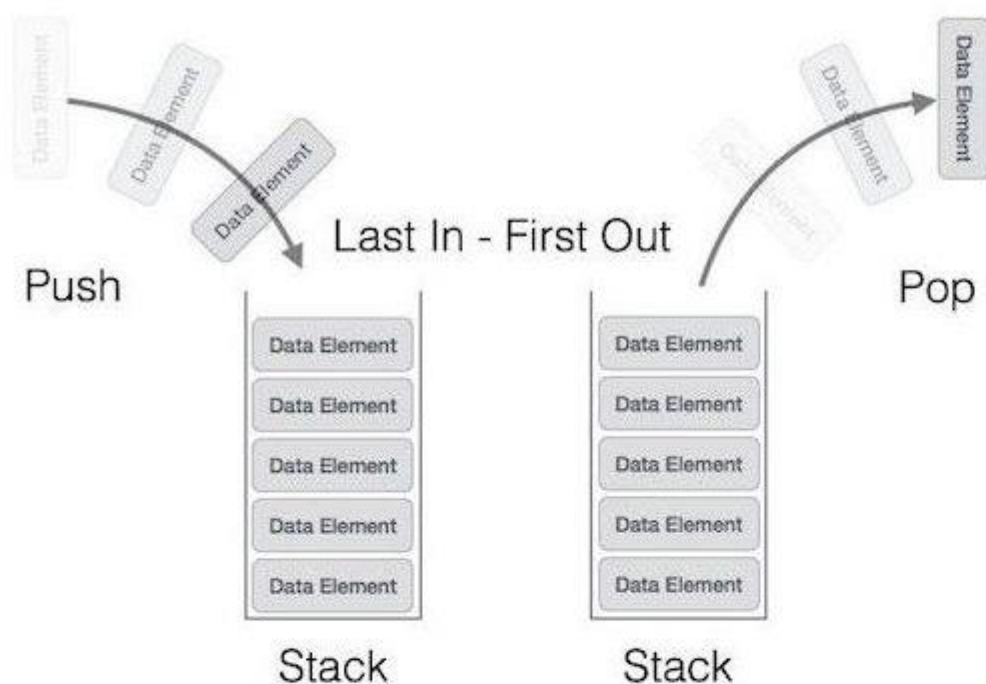


Рисунок 2.1. Структура стеку

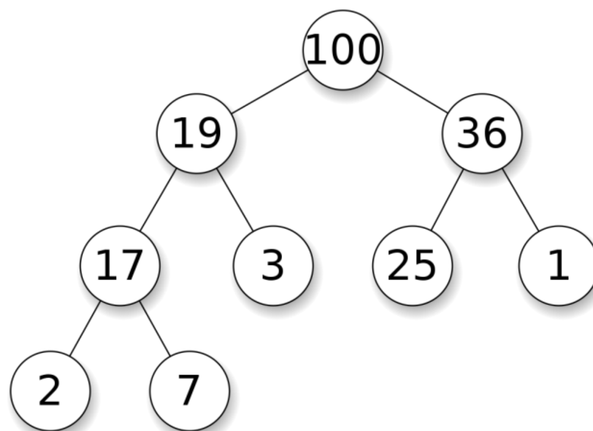
Купа - це сховище пам'яті, також розташоване в ОЗУ, яке дозволяє динамічне виділення пам'яті і не працює за принципом стека: це просто склад для змінних.

Коли виділяємо в купі ділянку пам'яті для зберігання змінної, до неї можна звернутися не тільки в потоці, але й у всьому додатку. Саме так визначаються глобальні змінні. Після завершення програми всі виділені ділянки пам'яті звільняються.

Розмір купи задається при запуску програми, але, на відміну стека, він обмежений лише фізично, і це дозволяє створювати динамічні змінні.

Взаємодія з купою відбувається за допомогою посилань, які зазвичай називають вказівниками — це змінні, чиї значення є адресами інших змінних. Створюючи вказівник, ми вказуємо на місце розташування пам'яті в купі, що задає початкове значення змінної і говорить програмі, де отримати доступ до цього значення.

У порівнянні зі стеком купа працює повільніше, оскільки змінні розкидані по пам'яті, а не сидять на верхівці стека. Некоректне керування пам'яттю в купі призводить до уповільнення її роботи.



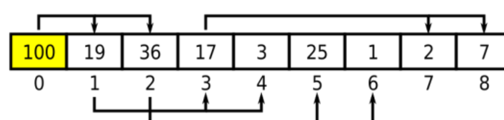


Рисунок 2.2. Приклад повної бінарної купи.

Отже, коротко підсумовуючи все зазначене вище, стек - це дуже швидке сховище пам'яті, що працює за принципом LIFO та кероване процесором. Але ці переваги призводять до обмеженого розміру стека та спеціального способу отримання значень. Купа дозволяє створювати динамічні та глобальні змінні - але керувати пам'яттю повинен або збирач сміття, або сам програміст, та й купа працює повільніше.

Як результат, можна зробити висновки про те, що структури будуть більш оптимальним варіантом застосування в реалізації методу Куайна – Маак-Класкі за рахунок менш витратного за ресурсами використання пам’яті.

Чим більше ви використовуватимете структури замість маленьких класів, тим менш витратним за ресурсами буде використання пам'яті.

2.2.2 Додавання пулу об’єктів

Розглянемо більш детально механізм додавання пулів, щоб зрозуміти його потенційні ефект та наслідки застосування.

Об'єктний пул - шаблон проектування, набір ініціалізованих і готових до використання об'єктів. Коли системі потрібен об'єкт, він не створюється, а береться з пула. Коли об'єкт більше не потрібен, то він не знищується, а повертається в пул.

Об'єктний пул застосовується для підвищення продуктивності, коли створення об'єкта на початку роботи і знищення його в кінці призводить до великих витрат. Особливо помітно підвищення продуктивності, коли об'єкти часто створюються-знищуються, але водночас існує лише невелика їхня кількість.

Якщо в пулі немає жодного вільного об'єкта, можлива одна з трьох стратегій:

- розширення пулу;
- відмова у створенні об'єкта, аварійна зупинка;
- у разі багатозадачної системи можна почекати, поки один з об'єктів не звільниться.

Проте даний метод має декілька пасток. Після того, як об'єкт повернено, він повинен повернутися до стану, придатного для подальшого використання. Якщо об'єкти після повернення в пул опиняються в неправильному чи невизначеному стані, така конструкція називається об'єктною.

Повторне використання об'єктів може призвести до витоку інформації. Якщо в об'єкті є секретні дані, після звільнення об'єкта цю інформацію треба затерти.

Отже, підсумовуючи вищесказане, пул об’єктів може допомогти запобігти надмірному виділенню пам’яті завдяки повторному використанню об’єктів програми. Проте варто приділити достатньо уваги тому, щоб об’єкти пулу поверталися до нього у належному для повторного використання стані.

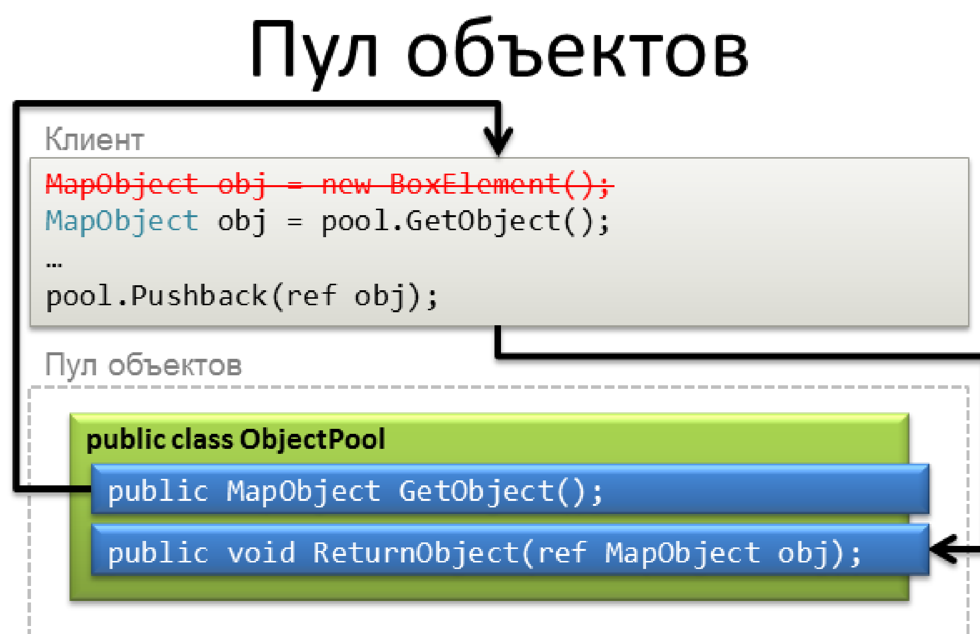


Рисунок 2.3. Приклад пулу об'єктів

2.2.3 Явний виклик методів Garbage Collector

Основною ідеєю є максимальна очистка пам'яті збирачем сміття перед кожною ітерацією всередині реалізації алгоритму Куайна – Мак-Класкі.

Розглянемо що собою являє Garbage Collector та в чому полягає суть його роботи.

GC (Garbage Collector) - високорівнева абстракція, яка позбавляє розробників необхідності дбати про звільнення керованої пам'яті.

У .NET складання сміття засноване на трасуванні.

Існує поняття кореневих елементів програми. Кореневим елементом (root) називається комірка в пам'яті, в якій міститься посилання на об'єкт, що розміщується в купі. Строго кажучи, кореневими можуть бути такі елементи:

- Посилання на глобальні об'єкти (хоча в C# вони не дозволені, але CIL-код дозволяє розміщувати глобальні об'єкти).
- Посилання на будь-які статичні об'єкти чи статичні поля.
- Посилання на локальні об'єкти в межах кодової бази програми.
- Посилання на параметри об'єкта, що передаються методу.
- Посилання на об'єкт, що очікує на фіналізацію.
- Будь-які регістри центрального процесора, які посилаються на об'єкт.

Під час процесу збирання сміття виконуюче середовище досліджуватиме об'єкти в купі, щоб визначити, чи є вони, як і раніше, досяжними (тобто кореневими) для програми. Для цього середовище CLR буде створювати графи об'єктів, що представляють всі доступні для програми об'єкти. Крім того, слід мати на увазі, що збирач сміття ніколи не створюватиме граф для одного і того ж об'єкта двічі, позбавляючи необхідності виконання підрахунку циклічних посилань, який характерний для програмування в середовищі COM.

Наприклад, у нас є керована купа з набором елементів A, B, C, D, E, F, G, I (рис. 2.3).

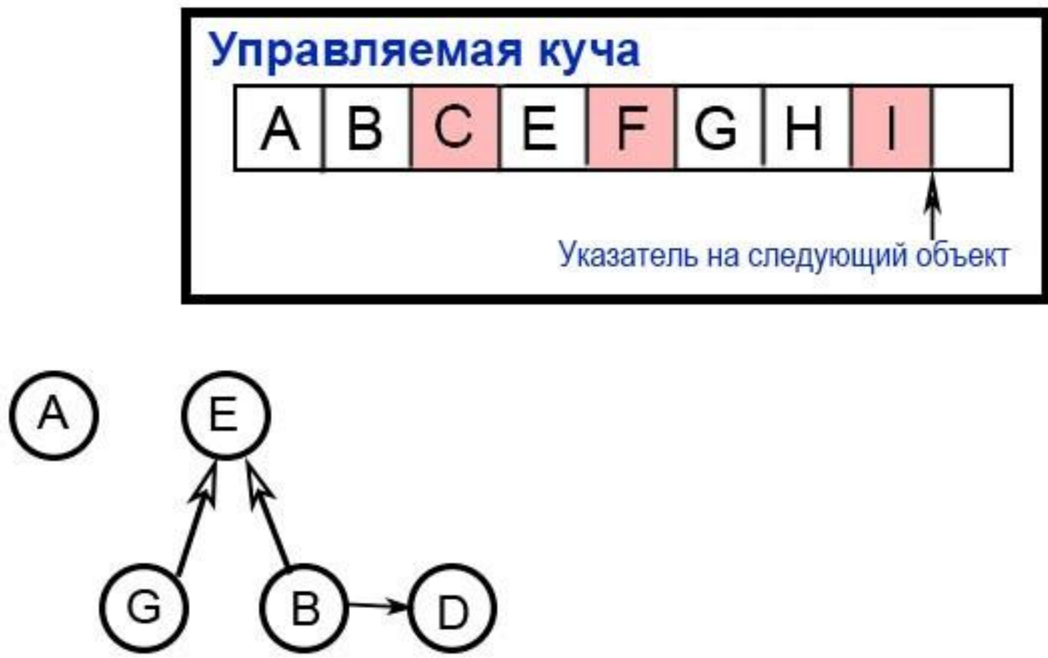


Рисунок 2.3. Приклад керованої купи

У цьому прикладі недоступними виявилися об'єкти C, F, I. Дані об'єкти будуть видалені з пам'яті, простір, що залишився в купі стискається, вказівник на наступний елемент налаштовується таким чином, щоб вказувати на наступний доступний ділянку пам'яті. Кінцевий результат буде наступним (рис 2.4):

Рисунок 2.4. Вигляд керованої купи після роботи Garbage Collector' а

Однак потрібно розуміти, що якби щоразу перевірявся кожен об'єкт, що знаходиться в купі, то це зайняло б масу часу і вся користь від автоматичного збирання сміття могла б витіснятися постійно гальмуючими додатками. Для оптимізації цього процесу був придуманий механізм «поколінь».

Кожен об'єкт у купі відповідає одному з трьох поколінь(рис. 2.5):

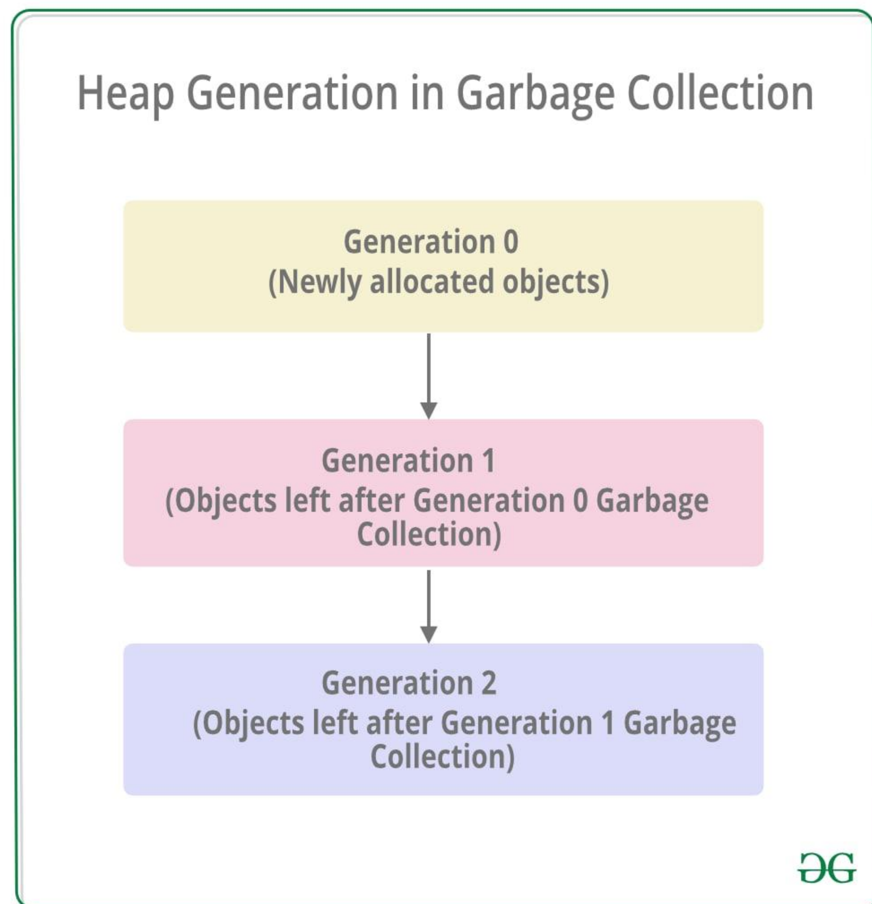


Рисунок 2.5. Покоління Garbage Collector' а

- Покоління 0: Новий розміщений у купі об'єкт, який ще ніколи не позначався як сміття, що підлягає складання.
- Покоління 1: Об'єкт, який вже пережив процес складання сміття.
- Покоління 2: Об'єкт, який пережив більше одного процесу збирання сміття.

Ідея «поколінь» проста: чим довше об'єкт знаходиться в купі, тим вища ймовірність того, що він у ній і залишиться.

Спочатку збирач сміття аналізує всі об'єкти покоління 0, якщо після видалення всіх непотрібних об'єктів залишається достатня кількість пам'яті, то всі елементи, що залишилися, підвищуються до покоління 1. Якщо всі об'єкти покоління 0 перевірені, але як і раніше потрібна пам'ять, то починається перевірка на досяжність і видалення об'єктів з покоління 1, уцілілі об'єкти покоління 1 підвищуються до покоління 2. Якщо ж збиральнику сміття як і раніше потрібна пам'ять, то перевірку на досяжність і видалення починають проходити об'єкти покоління 2. Об'єкти покоління 2, яким вдалося пережити складання сміття, як і раніше залишаються об'єктами 2-го покоління тому що це старше покоління.

Процес збирання сміття запускається у наступних випадках:

- Заповнено покоління 0;
- Виклик із коду GC.Collect;
- Операційна система повідомляє, що недостатньо пам'яті;

Існує можливість програмного управління збирачем сміття. І тому існує тип System.GC. Щоб змусити збирача сміття провести прибирання, достатньо викликати метод Collect, доступний у класі GC. При виклику також можна вказати номер покоління, в якому необхідно прибрати сміття.

Отже, в результаті застосування методів для примусового збирання сміття, потенційно можна отримати більшу кількість пам’яті для роботи програми.

Висновки до другого розділу

При реалізації методу Куайна – Мак-Класкі в середовищі Visual Studio на мові програмування C# нами були виявлені наступні характерні недоліки та проблеми при роботі з даним методом:

- Низька швидкість роботи програмного додатку;
- При великій кількості змінних програма завершується помилкою через нестачу пам’яті.

Оскільки, низька швидкість роботи методу є вже досить відомим недоліком, основна увага в даній роботі була зосереджена на виявлені шляхів вирішення проблеми нестачі пам’яті. Було розглянуто декілька варіантів вирішення зазначеної раніше проблеми:

- Заміна класів мови C#, що використовуються в реалізації методу Куайна – Мак-Класкі, на аналогічні їм структури;
- Додавання пулу об’єктів з метою повторного використання об’єктів, а не створення та виділення пам’яті під нові;
- Явне використання методів збирача сміття (Garbage Collector) з метою максимальної очистки пам’яті.

Проведемо детальний аналіз кожного з методів, оцінимо доцільність їх використання при реалізації методу Куайна – Мак-Класкі, а також проаналізуємо отримані результати в наступному розділі.

РОЗДІЛ 3.
ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Програмна реалізація методу Куайна - Мак-Класкі

Перш ніж розпочати застосування потенційних методів вирішення проблем з виділенням пам'яті, розглянемо саму реалізацію методу Куайна – Мак-Класкі.

Відповідно до теорії метод Мак-Класкі складається з двох основних етапів:

- Знаходження всіх простих терм ЛФ, використовуючи правило (закони) склеювання:
  a) (A & B) ∨ (A & !B) ≡ A;
  b) (A ∨ B) & (A ∨ !B) ≡ A;
  де & — операція логічного «І»; ∨ — операція логічного «АБО»;! – операція логічного заперечення «НЕ».
- Мінімізація кількості простих терм отриманої множини, як завдання знаходження оптимального покриття множини підмножинами різної довжини.

В створеній програмі клас Quine_McCluskey є власне реалізацією цього алгоритму, якому допомагають інші класи та інтерфейси:

- Dnf,
- TreeNodeBase,
- TreeNodeMiddle,
- TreeNodeEnd,
- TreeFuncTerm.

Для запуска процесу оптимізації необхідно викликати один з перевантажених методів Start.

У пункті 1 відбувається багаторазовий попарний перебір терм з метою виявлення можливості їх склейки. Два терми склеюються, якщо вони відрізняються один від одного значеннями тільки в одній з позицій: в одного стоїть '0', а в іншого - '1'. Вихідні терми, що склеюються, виключаються з подальшої роботи, а замість них далі на наступній ітерації п. 1 алгоритму розглядається нова дочірня терма, у якої на позиції змінних батьківських терм, що різняться, стоїть знак '*'. Отже, двійковий алфавіт вхідних вихідних терм всередині алгоритму розширюється до троїчного: '0', '1', '*'. Слід зазначити, що у реалізації замість масивів символів використовуються байтові масиви, у яких символам '0', '1' і '*' відповідають значення байт 0, 1 і 2.

Пункт 1 у класі Quine_McCluskey реалізований у процедурі LogicFuncMinimize двома принципово різними способами. Перший спосіб, заснований на 64 бітних цілих хеш-кодах і пошуку в .NET словнику Dictionary<UInt64,…>, працює при кількості вхідних змінних булівської функції менше або дорівнює 40, а другий, заснований на пошуку в троїчному дереві, включається при більшому їх кількості. Таке роздвоєння обумовлено тим, що перший спосіб працює дещо швидше, ніж другий, тому його використання пріоритетніше, але він працює коректно тільки при кількості вхідних змінних до 40, а при більшій їх кількості виходить за 64 біт розрядну сітку цілого хеш-коду, використовуваного для роботи алгоритму. Дійсно, тому що в термах всередині алгоритму застосовується трійкова логіка, то при кількості вхідних змінних дорівнює 41 максимальне значення хеш-коду 3^{41} вже перевищує максимальне значення (2^{64}), яке можна записати у 64 бітну змінну.

Другий спосіб роботи п. 1, що працює при кількості змінних у вхідних термах більше 40, заснований на троїчному пошуковому дереві терм

TreeFuncTerm. Проміжні вузли дерева реалізовані за допомогою класу TreeNodeMiddle. Кожен з них може посилатися щонайбільше на три наступні вузли залежно від того чи були в дерево додані відповідні терми. Усі кінцеві вузли є екземплярами класу TreeNodeEnd, глибина до яких від кореня у всіх однакова і дорівнює кількості вхідних змінних. Кожен кінцевий вузол також має посилання на інший вузол TreeNodeEnd, який був доданий до нього, таким чином реалізуючи односпрямований зв'язаний список. Такий список використовується для швидкого перебору всіх кінцевих вузлів пошукового дерева в процесі їх склеювання.

Якщо в п. 1 для терма не знаходиться іншого, що відрізняється від нього тільки в одній позиції терма, тобто терм ні з ким не склеюється, то він вважається одним з результатів роботи п. 1 алгоритму, виключається з подальшої роботи в ньому і надходить на вхід пункту 2 роботи алгоритму, реалізований у процедурі ReduceRedundancyTerms.

Відкидання надлишкових терм у ReduceRedundancyTerms здійснюється за допомогою алгоритму наближеного розв'язання задачі про покриття безлічі підмножин змінної довжини. Покриття, близьке до найкоротшого, дає алгоритм перетворення таблиці покриттів (ТП), заснований на методі “мінімальний стовпець–максимальний рядок”.

Логіка його роботи полягає в наступному:

1. Вихідна таблиця вважається поточною ТП, що перетворюється, множина рядків покриттів - порожня;
2. У поточній таблиці виділяється стовпець із найменшим числом одиниць. Серед рядків, що містять одиниці в цьому стовпці, виділяється один із найбільшим числом одиниць. Цей рядок включається до покриття, поточна таблиця скорочується викреслюванням всіх стовпців, у яких вибраний рядок має одиницю;

3. Якщо таблиці є не викреслені стовпці, то виконується п. 2, інакше – покриття побудовано.

При підрахунку числа одиниць у рядку враховуються одиниці у невикреслених стовпцях.

Для перевірки роботи алгоритму використовують дві тестові функції. Перша процедура дозволяє для набору вхідних змінних кількістю N сформувати сукупність терм кількістю 2^N, обчислити випадкове значення ЛФ для кожного терму, провести мінімізацію на основі тих термів, для яких відповідне їм значення функції дорівнювало TRUE, і вивантажити їх у разі необхідності в файл. Після цього проводиться перевірка правильності роботи мінімізованої форми функції шляхом обчислення її значення для кожного терму та порівняння з вихідним.

Друга тестова процедура дозволяє завантажити з файлу набори, записані попередньою тестовою функцією TestQuineMcCluskeyRandom, для яких булевська процедура повинна приймати значення TRUE і провести перевірку правильності роботи мінімізованої функції.

3.2 Застосування запропонованих методів вирішення проблеми реалізації методу Куайна - Мак-Класкі на платформі .NET

3.2.1 Результати заміни класів вузлів троїчного дерева на структури

Перевіримо результати застосування ідей даного методу оптимізації роботи, замінивши класи TreeNodeEnd та TreeNodeMiddle на аналогічні їм структури мови C#.

В теорії така заміна могла б забезпечити нам деяку відповідну економію пам’яті. Проте, дана ідея зазнала своєї поразки ще на підготовчому етапі.

Причиною цього є наступне: дана реалізація методу Куайна – Мак-Класкі використовує в своєму коді посилання на TreeNodeEnd. І хоч є можливість застосовувати ключове слово ref для отримання посилання на структуру, головною перешкодою стає те, що ці самі об’єкти TreeNodeEnd можуть мати значення NULL, чого структури в свою чергу не допускають. Оскільки структури належать до значимих типів, а не до посилальних, вони апріорі позбавлені можливості мати в собі значення NULL.

Отже, в результаті проведення цього дослідження, можна впевнено сказати, що варіант заміни класів на структури є недієвим. Більш того, він є повністю незастосовним для даної реалізації методу Куайна – Мак-Класкі в виду вбудованих властивостей структур.

3.2.2 Результати додавання пулу об’єктів до програми

Даний підхід, так само як і попередній, мав би забезпечити нам майже гарантовано більшу кількість пам'яті для роботи програми. Проте, беручи, на жаль, поганий приклад з попереднього підходу, дана ідея також стає провальною ще до початку її практичної реалізації та тестування. Розглянемо що стало причиною такого результату.

Перш за все, згадаємо суть пулу об’єктів: можна не створювати нові об’єкти, а використовувати ті, що знаходяться в пулі. Тобто, іншими словами, є можливість повторно використовувати один й той самий об’єкт пулу. Але варто пам’ятати про те, що для досягнення ефективності від ідеї повторного використання об’єктів необхідно повертати цей об’єкт назад до пулу. Саме цей нюанс і стає на заваді використання пулу об’єктів для представленій в даній роботі реалізації методу Куайна – Мак-Класкі. Нестача пам’яті відбувається в одному місці однієї процедури, де масово створюються об’єкти типу TreeNodeEnd. Яне має змоги повторно використати один й той самий об’єкт з

пулу через те, що на той момент він ще використовується для подальших розрахунків і не повернувся назад до пулу об'єктів. Отже, в такому випадку, застосування пулу об'єктів не дасть ніякої ефективності.

3.2.3 Результати явного виклику методів Garbage Collector

На відміну від двох попередніх ідей щодо покращення роботи програми, даний метод є теоретично застосовним і ніщо не впливає на можливість примусового виклику методів для збирання сміття.

Отже, для перевірки ефективності роботи даної ідеї нам, перш за все необхідно додати явний виклик методу GC.Collect().

Надмірне використання пам'яті відбувається всередині п.1 алгоритму Куайна – Мак-Класкі. Тож, додамо явний виклик збирача сміття в код функції LogicFuncMinimize.

Протестуємо отриману програму та проаналізуємо її результати. Тестування програми буде відбуватись на двох різних машинах, які відрізняються ступенем «засміченості» пам'яті. Таким чином буде можливість подивитись роботу програми в різних умовах. Також, буде перевірена доцільність використання ідеї збирача сміття та її залежність від машини, на якій запускається програма.

Параметри комп'ютерів, на яких буде проводитися тестування:

1. Комп'ютер №1:
   - процесор: Intel Core i5-11400F;
   - кількість ядер: 6;
   - кількість логічних процесорів: 12;
   - встановлена оперативна пам'ять: 16 ГБ.

2. Комп’ютер №2:
   - процесор: Intel Core i3-5005U;
   - кількість ядер: 2;
   - кількість логічних процесорів: 4;
   - встановлена оперативна пам’ять: 4 ГБ.

Отримані результати тестування:

Комп’ютер №1:

- без використання методу GC.Collect():

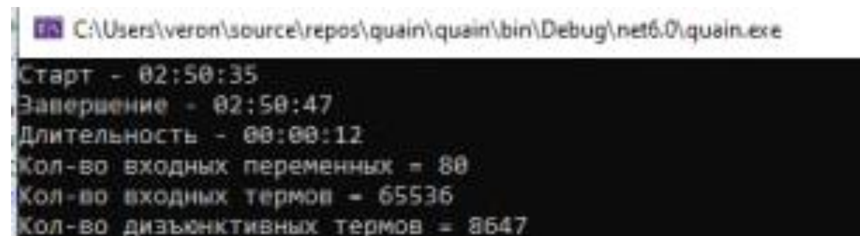


Рисунок 3.1. Результат роботи програми на комп'ютері №1 без використання методу GC.Collect() (1)

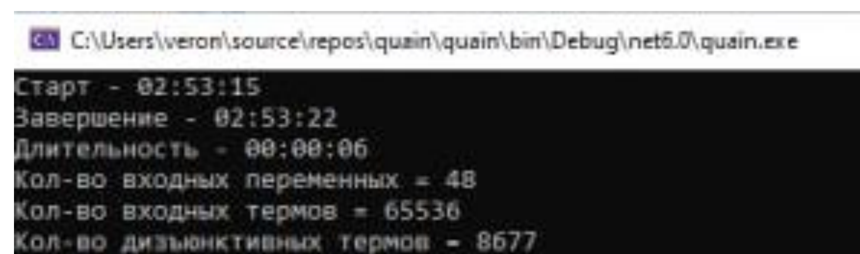


Рисунок 3.2. Результат роботи програми на комп'ютері №1 без використання методу GC.Collect() (2)

- з використанням методу GC.Collect():

```
C:\Users\veron\source\repos\quain\quain\bin\Debug\net6.0\quain.exe
Старт - 02:51:31
Завершение - 02:51:43
Длительность - 00:00:11
Кол-во входных переменных = 80
Кол-во входных термов = 65536
Кол-во дизъюнктивных термов = 8705
```

Рисунок 3.3. Результат роботи програми на комп'ютері №1 з використанням методу GC.Collect() (1)

```
C:\Users\veron\source\repos\quain\quain\bin\Debug\net6.0\quain.exe
Старт - 02:53:57
Завершение - 02:54:04
Длительность - 00:00:07
Кол-во входных переменных = 48
Кол-во входных термов = 65536
Кол-во дизъюнктивных термов = 8623
```

Рисунок 3.4. Результат роботи програми на комп'ютері №1 з використанням методу GC.Collect() (2)

Комп’ютер №2:

- без використання методу GC.Collect():

```
C:\Users\Comp\source\repos\ConsoleApp4\ConsoleAp
Старт - 02:21:15
Завершение - 02:22:03
Длительность - 00:00:47
Кол-во входных переменных = 48
Кол-во входных термов = 65536
Кол-во дизъюнктивных термов = 8631
```

Рисунок 3.5. Результат роботи програми на комп'ютері №2 без використання методу GC.Collect() (1)

```
C:\Users\Comp\source\repos\ConsoleApp4\Consol
Старт - 02:34:44
Завершение - 02:35:55
Длительность - 00:01:11
Кол-во входных переменных = 80
Кол-во входных термов = 65536
Кол-во дизъюнктивных термов = 8676
```

Рисунок 3.6. Результат роботи програми на комп'ютері №2 без використання методу GC.Collect() (2)

- з використанням методу GC.Collect():

```
C:\Users\Comp\source\repos\ConsoleApp4\Consol
Старт - 02:25:29
Завершение - 02:26:00
Длительность - 00:00:30
Кол-во входных переменных = 48
Кол-во входных термов = 65536
Кол-во дизъюнктивных термов = 8692
```

Рисунок 5. Результат роботи програми на комп'ютері №2 з використанням методу GC.Collect() (1)

```
C:\Users\Comp\source\repos\ConsoleApp4\Consol
Старт - 02:40:04
Завершение - 02:41:04
Длительность - 00:01:00
Кол-во входных переменных = 80
Кол-во входных термов = 65536
Кол-во дизъюнктивных термов = 8626
```

Рисунок 6. Результат роботи програми на комп'ютері №2 з використанням методу GC.Collect() (2)

Після проведення тестування, отримано наступні результати. Явний виклик методу збирача сміття дійсно допомагає у вирішенні проблеми нестачі пам’яті при роботі даної реалізації методу Куайна – Мак-Класкі. Є два можливі варіанти чому:

- збирач сміття таки збирає на кожній ітерації деякі об’єкти, тим самим звільняючи пам’ять;
- GC.Collect() дефрагментує пам'ять, завдяки чому туди може «влізти» більше об’єктів типу TreeNodeEnd.

Не менш важливим та, навіть, неочікуваним результатом явного застосування методу збирача сміття стало зменшення часу роботи програми, хоча, на перший погляд, воно мало б навпаки збільшитись за рахунок витрати часу на виклик методу та самий збір сміття. Особливо помітний цей «феномен» при тестуванні програми на комп’ютері №2. Поясненням виникнення цього результату може бути те, що вивільнення пам’яті збирачем сміття зменшує її дефрагментацію в купі, що, в свою чергу, позитивно впливає на швидкість пошуку вільних ділянок при подальшому її масовому виділенні.

Проте, варто зазначити те, що суттєвий вплив на швидкість роботи програми збирач сміття має саме на комп’ютері, що має більш «засмітнену» та, можливо навіть, більш «пошкоджену» пам’ять. Але, навіть не зважаючи на даний фактор, можна впевнено сказати про доцільність використання явного виклику методу збирача сміття для оптимізації роботи алгоритму Куайна – Мак-Класкі.

Висновки до третього розділу

Був програмно реалізований метод Куайна – Мак-Класкі.

Проаналізовано запропоновані раніше методи для оптимізації роботи даного методу. Результатами аналізу є наступне:

- Ідея застосування структур є незастосовною для даного методу через свою вбудовану властивість – неможливість мати в собі значення NULL, при тому що деякі елементи реалізації можуть потенційно містити в собі таке значення.
- Застосування пулу об’єктів є недоцільним для даного методу. Головна ідея пулу об’єктів полягає в повторному використанні об’єктів програми, та нам при цьому необхідно повертати їх назад до пулу. Проте, проблема нестачі пам’яті в реалізації методу Куайна – Мак-Класкі відбувається на етапі масового створення об’єктів, які будуть використовуватись на далі і, виходячи з цього, вони ще не можуть бути повернуті до пулу об’єктів для повторного використання.
- Явний виклик методів збирача сміття є застосовним і ніщо не заважає його використовувати.

Після аналізу можливості застосування кожного з варіантів покращення роботи методу Куайна – Мак-Класкі ми практично реалізували та протестували роботу програму з застосуванням явного виклику методів збирача сміття.

Тестування отриманої програми відбувалось на двох різних машинах з різними технічними покажчиками.

В результаті отримано висновок про те, що ідея застосування збирача сміття дійсно допомагає вирішити проблему з нестачею пам’яті. Проте, в ході тестування було також виявлено ще один важливий результат. При застосуванні

методів збирача сміття в деяких випадках була збільшена швидкість роботи програми, хоча очікувалось зворотне – зменшення швидкості роботи за рахунок витрат часу на виклик методу збирання сміття, а також його виконання. Особливо помітним цей ефект виявився при тестуванні програми на комп’ютері з більш низькими технологічними показниками, а також який має більш «засмічену» пам’ять. Цей ефект можна пояснити тим, що збирач сміття зменшує дефрагментацію пам’яті, що в свою чергу зменшує час на пошук вільних ділянок пам’яті для нових об’єктів.

РОЗДІЛ 4.
ЕКОНОМІКА

4.1. Визначення трудомісткості розробки програмного забезпечення

Одним з головних етапів при розробці ПЗ є визначення трудомісткості та розрахунок витрат на створення програмного продукту. В даному розділі буде продемонстровано приклад розрахунку витрат на розробку програми для розпізнавання неістинності міркувань.

Початкові дані:

- годинна заробітна плата програміста, грн / год — 200;
- вартість машино-години ЕОМ, грн / год — 40.

Нормування роботи в процесі створення ПЗ істотно ускладнюється в силу творчого характеру роботи програміста, тому трудомісткість розробки ПЗ може бути розрахована на основі системних моделей з різною точністю оцінки.

Трудомісткість розраховується за формулою:

$$t = t_u + t_a + t_n + t_{відл} + t_д ,$$

t_o – витрати праці на підготовку і опис поставленого завдання (50 год.);

t_u – витрати праці на дослідження алгоритму вирішення задач;

t_a – витрати праці на розробку блок-схем алгоритму;

$t_п$ – витрати праці на програмування за готовим блок-схемою;

$t_{відл}$ – витрати праці на відладку програм на ПЕОМ;

$t_д$ -– витрати праці на підготовку документації.

Сукупні витрати праці визначаються виходячи з умовного числа операторів у розроблюваному ПЗ.

Розрахунок очікуваних витрат праці:

$$Q = q \times C \times (1+ p),\ \text{людино-годин},$$

де q – передбачуване число операторів;

c – коефіцієнт складності програми;

p – коефіцієнт корекції програми в ході її розробки.

Наприклад, q буде дорівнювати 1050, c – 1,3, p – 0,1.

$$Q=1050\times 1,3\times(1+0,1) = 1500\ \text{людино-годин},$$

Витрати праці на вивчення опису завдання визначаються з урахуванням уточнення опису і кваліфікації програміста за формулою:

$$t_u = \frac{QB}{(75 \dots 85)K},$$

де B – коефіцієнт збільшення витрат праці (внаслідок неповного опису завдання, B= 1,2…..1,5);

K – Коефіцієнт кваліфікації програміста, який визначається в залежності від стажу роботи за фахом (якщо стаж роботи менший 2 років, то K = 1,2).

Витрати праці на розробку алгоритму рішення задачі:

$$t_u = 1500 * 1,3 / (80 * 1,2) = 20,3\ \text{людино-годин},$$

Витрати на складання програми по готовій блок-схемі:

$$t_a = \frac{Q}{(20 \dots 25)K}\ \text{людино} - \text{годин},$$

$$t_a = 1500 / 23 * 1,2 = 78,2\ \text{людино-годин},$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20 \ldots 25)K} \text{ людино} - \text{годин},$$

t_n = 15 /23 * 1,2 = 78,2 людино-годин,

Витрати праці на налагодження програми на ЕОМ:

$$t_{\text{відл}} = \frac{Q}{(4 \ldots 25)K} \text{ людино} - \text{годин},$$

$t_{\text{відл}}$ = 1500 / 4 * 1,2 = 450 людино-годин

Витрати праці на підготовку документації:

$$t_{\text{д}} = t_{\text{др}} + t_{\text{до}} \text{ людино} - \text{годин},$$

де $t_{\text{др}}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{\text{др}} = \frac{Q}{(15 \ldots 20)K} \text{ людино} - \text{годин},$$

$t_{\text{др}}$ = 1500 / 17 * 1,2 = 105 людино-годин,

$t_{\text{до}}$ – трудомісткість редагування, печатки й оформлення документації

$$t_{\text{до}} = 0.75 + t_{\text{др}} \text{ людино} - \text{годин},$$

$t_{\text{до}}$ = 0,75 * 105 = 79,4 людино-годин,

$t_{\text{д}}$ = 105 + 79,4 = 184,4 людино-годин,

Отримуємо трудомісткість розробки програмного забезпечення:

t = 50 +20,3 + 78,2 +78,2 + 450 + 184,4 = 861,1 людино-годин.

4.2. Розрахунок витрат на створення програмного забезпечення

Витрати на створення програмного забезпечення $К_{пз}$ включають витрати на заробітну плату розробників програми ($З_{зп}$) і витрати машинного часу, необхідного для налагодження програми на ЕОМ ($З_{мв}$).

$$К_{пз} = З_{зп} + З_{мв}$$

$$З_{зп} = t * С_{пр}$$

де t – загальна трудомісткість розробки ПЗ;

$С_{пр}$ – середня годинна заробітна плата програміста.

$$З_{зп} = 861{,}1 * 200 = 172220 \text{ грн}$$

Витрати машинного часу, необхідного для налагодження програми на ЕОМ ($З_{мв}$) визначаються за формулою:

$$З_{мв} = t_{відл} * С_{мч}$$

где $t_{відл}$ – трудомісткість налагодження програми на ЕОМ;

$С_{мч}$ – вартість машино-години ЕОМ.

$$З_{мв} = 450 \times 40 = 18000 \text{ грн.}$$

Таким чином витрати на створення програмного забезпечення, складуть:

$$К_{пз} = 172220 + 18000 = 190220 \text{ грн.}$$

Очікувана тривалість розробки ПЗ:

$$T = \frac{t}{B_k \cdot F_p}$$

где B_k – число розробників;

F_p– місячний фонд робочого часу (при 40-ка годинному робочому тижні F_p = 176 годин).

T = 861,1 / 1 * 176 = 5 місяців.

4.3. Маркетингові дослідження

Для розрахунку маркетингової частини на даний момент жодна з аналогічних існуючих програм з паралельними обчисленнями не задовольняє вимогам, тому що їх вихідний код невідомий, а тому складність розробки неможливо порівняти.

4.4. Економічна ефективність

Економічний ефект від впровадження ПЗ, яке використовує паралельні обчислення, очікується позитивним так як від швидкості роботи такого роду додатків залежить маса високонавантажених розрахунків у проектувальній та будівельний галузі.

ВИСНОВКИ

У процесі дослідження були отримані наступні результати. Розглянуто 4 основні методи створення систем штучного інтелекту серед яких:

1. Логічний підхід. Основою для вивчення логічного підходу слугує алгебра логіки. Свого подальшого розвитку алгебра логіки отримала у вигляді числення предикатів — в якому вона розширена за рахунок введення предметних символів, відношень між ними.
2. Структурний підхід. Спроби побудови ШІ шляхом моделювання структури людського мозку. Базується на перцептронах. Головною моделюючою структурною одиницею в перцептронах є нейрон.
3. Еволюційний підхід. Під час побудови системи ШІ за даним методом основну увагу зосереджують на побудові початкової моделі і правилах, за якими вона може змінюватися (еволюціонувати).
4. Імітаційний підхід. Цей підхід є класичним для кібернетики з одним із її базових понять чорний ящик. Об'єкт, поведінка якого імітується, якраз і являє собою «чорний ящик». Для нас не важливо, які моделі у нього всередині і як він діє, головне, щоби наша модель в аналогічних ситуаціях поводила себе без змін.

В ході даного дослідження основна увага була зосереджена на численні висловлень, його особливостях, а також методах, за якими можна перевірити виводимість інформації.

Серед методів та алгоритмів перевірки виводимості формул можна виділити наступні:

- Алгоритм істинних таблиць (АІТ);

- Метод резолюцій;
- Алгоритм Куайна, який був згодом покращений Едвардом Мак-Класкі.

Для подальших досліджень був взятий метод перевірки виводимості формул Куайна – Мак-Класкі. Алгоритм його роботи полягає в наступному:

1. Терми (кон'юнктивні у разі СДНФ та диз'юнктивні у разі СКНФ), на яких визначено функцію алгебри логіки (ФАЛ) записуються у вигляді їх двійкових еквівалентів;
2. Ці еквіваленти розбиваються на групи, до кожної групи входять еквіваленти з рівною кількістю одиниць (нулів);
3. Проводиться попарне порівняння еквівалентів (термів) у сусідніх групах з метою формування термів нижчих рангів;
4. Складається таблиця, заголовком рядків у якій є вихідні терми, а заголовком стовпців - терми низьких рангів;
5. Розставляються мітки, що відбивають поглинання термів вищих рангів (вихідних термів) і далі мінімізація проводиться у спосіб Куайна.

При проведені дослідження були виявлені характерні недоліки при реалізації даного алгоритму:

- Низька швидкість роботи програмного додатку;
- При великій кількості змінних програма завершується помилкою через нестачу пам’яті.

Оскільки, низька швидкість роботи методу є вже досить відомим недоліком, основним напрямком дослідження стала оптимізація роботи алгоритму Куайна – Мак-Класкі саме з погляду на усунення проблем, пов’язаних

з нестачею пам’яті при наявності в програмі великої кількості змінних. Розглянуто декілька варіантів вирішення проблеми:

- Заміна класів мови C#, що використовуються в реалізації методу Куайна – Мак-Класкі, на аналогічні їм структури;
- Додавання пулу об’єктів з метою повторного використання об’єктів, а не створення та виділення пам’яті під нові;
- Явне використання методів збирача сміття (Garbage Collector) з метою максимальної очистки пам’яті.

Після того, як було програмно реалізовано метод Куайна – Мак-Класкі за допомогою інструментів мови C# і проведено аналіз можливості застосування запропонованих раніше варіантів вирішення проблеми нестачі пам’яті, були зроблені наступні висновки:

- Ідея застосування структур є незастосовною для даного методу через свою вбудовану властивість – неможливість мати в собі значення NULL, при тому що деякі елементи реалізації можуть потенційно містити в собі таке значення.
- Застосування пулу об’єктів є недоцільним для даного методу. Головна ідея пулу об’єктів полягає в повторному використанні об’єктів програми, та нам при цьому необхідно повертати їх назад до пулу. Проте, проблема нестачі пам’яті в реалізації методу Куайна – Мак-Класкі відбувається на етапі масового створення об’єктів, які будуть використовуватись на далі і, виходячи з цього, вони ще не можуть бути повернуті до пулу об’єктів для повторного використання.
- Явний виклик методів збирача сміття є застосовним і ніщо не заважає його використовувати.

Було практично реалізовано та протестовано роботу програми з застосуванням явного виклику методів збирача сміття. Тестування отриманої програми відбувалось на двох різних машинах з різними технічними покажчиками.

В результаті отримано висновок про те, що ідея застосування збирача сміття дійсно допомагає вирішити проблему з нестачею пам’яті. Проте, в ході тестування було також виявлено ще один важливий результат. При застосуванні методів збирача сміття в деяких випадках була збільшена швидкість роботи програми, хоча очікувалось зворотне – зменшення швидкості роботи за рахунок витрат часу на виклик методу збирання сміття, а також його виконання. Особливо помітним цей ефект виявився при тестуванні програми на комп’ютері з більш низькими технологічними показниками, а також який має більш «засмічену» пам’ять. Цей ефект можна пояснити тим, що збирач сміття зменшує дефрагментацію пам’яті, що в свою чергу зменшує час на пошук вільних ділянок пам’яті для нових об’єктів.

Узагальнення отриманого практичного досвіду буде корисно для подальших досліджень, пов’язаних з оптимізацією роботи методів перевірки виводимості формул, зокрема методу Куайна – Мак-Класкі.

Проведене дослідження буде корисно для науковців, розробників, аспірантів і студентів, що спеціалізуються або цікавляться проблематикою методів числення висловлень та їх програмної реалізації.

ПЕРЕЛІК ВИКОРИСТАННИХ ДЖЕРЕЛ

1. ISO/IEC TR 24028:2020 Information technology - Artificial intelligence - Overview of trustworthiness in artificial intelligence. International Organization for Standardization and International Electrotechnical Commissio (англ.). May 2020.
2. Миркес Е. М., Нейрокомп'ютер. Проект стандарту.- Новосибірськ: Наука, Сибірська видавнича фірма РАН, 1999 .- 337 с. ISBN 5-02-031409-9 (Глава 9: «Контрастер»)
3. McCulloch, Warren S.; Pitts, Walter (December 1943). A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics (en) 5 (4): 115–133.
4. Штучний інтелект [https://uk.wikipedia.org/wiki/%D0%A8%D1%82%D1%83%D1%87%D0%BD%D0%B8%D0%B9_%D1%96%D0%BD%D1%82%D0%B5%D0%BB%D0%B5%D0%BA%D1%82]
5. Логічне програмування [https://uk.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F]
6. Прийма С.М. Математична логіка і теорія алгоритмів: Навчальний посібник – Мелітополь: ТОВ „Видавничий будинок ММД", 2008. - 134 с.
7. Гасяк О.С. Формальна логіка : короткий словник-довідник. – Чернівці : Чернівецький нац. ун-т, 2014. – 200 с.
8. Логічні числення // Філософський енциклопедичний словник / В. І. Шинкарук (гол. редкол.) та ін. — Київ : Інститут філософії імені Григорія Сковороди НАН України : Абрис, 2002. — 742 с. — 1000 екз. — ББК 87я2. — ISBN 966-531-128-X.
9. Числення висловлювань // ФЕС, с.714

10. Матвієнко М.П., Шаповалов С.П. Математична логіка та теорія алгоритмів. Навчальний посібник. – К.: Видавництво Ліра-К, 2015. – 212 с.
11. Числення предикатів // ФЕС, с.714
12. Гільберт Д., Аккерман В. Основи теоретичної логіки. М., 1947
13. Кліні С. К. Введення в метаматематику. М., 1957
14. Мендельсон Э. Введення в математичну логіку. М., 1976
15. Новіков П. С. Елементи математичної логіки. М., 1959
16. Черч А. Введення в математичну логіку, т. I. М. 1960
17. Філософський словник / за ред. В. І. Шинкарука. — 2-ге вид., переоб. і доп. — К. : Головна ред. УРЕ, 1986.
18. Enderton, H. B., A Mathematical Introduction to Logic. Harcourt/Academic Press 2002. ISBN 0-12-238452-0
19. A. G. Hamilton Logic for Mathematicians, Cambridge University Press, Cambridge UK 1978 ISBN 0-521-21838-1.
20. Обчислення висловлень. Перевірка виведення правильних висновків. Алгоритм Квайна. Правило резолюцій. [https://studfile.net/preview/413969/]
21. J. Alan Robinson (1965), A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM (JACM), Volume 12, Issue 1, pp. 23-41.
22. Leitsch, Alexander (1997). The Resolution Calculus. Springer-Verlag.
23. Gallier, Jean H. (1986). Logic for Computer Science: Foundations of Automatic Theorem Proving. Harper & Row Publishers.
24. Чень Ч., Ли Р. Глава 5. Метод резолюцій // Математична логіка і автоматичне доведення теорем = Chin-Liang Chang; Richard Char-Tung Lee (1973). Symbolic Logic and Mechanical Theorem Proving. Academic Press. — М.: «Наука», 1983. — С. 358.
25. Гуц А.К. Глава 1.3. Метод резолюцій // Математична логіка и теорія алгоритмів. — Омск: Наследие. Диалог-Сибирь, 2003. — С. 108.
26. Нільсон Н. Дж. Принципи штучного інтелекту. — М., 1985.
27. Рассел С., Норвіг П. Штучний інтелект: сучасний підхід = Artificial Intelligence: a Modern Approach. — М.: Вільямс, 2006.

28. «Прикладна теорія цифрових автоматів», Київ «Вища Школа» 1987, К. Г. Самофалов, А. М. Романкевич, В. Н. Валуйский, Ю. С. Каневский, М. М. Пиневич.

29. Метод Куайна — Мак-Класкі [https://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9A%D1%83%D0%B0%D0%B9%D0%BD%D0%B0_%E2%80%94_%D0%9C%D0%B0%D0%BA-%D0%9A%D0%BB%D0%B0%D1%81%D0%BA%D1%96]

30. "Прикладна теорія цифрових автоматів" Київ "Вища Школа" 1987 К.Г. Самофалов, А.М. Романкевич, В.Н. Валуйський, Ю.С. Каневський, М.М. Пиневич с.(197 - 199).

31. Метод Квайна [https://studfile.net/preview/6308727/page:13/]

32. Метод Квайна – Мак-Класкі [https://studfile.net/preview/6308727/page:14/]

33. Masek, William J. (1979). Some NP-complete set covering problems. unpublished.

34. Czort, Sebastian Lukas Arne (1999). The complexity of minimizing disjunctive normal form formulas (Master's thesis). University of Aarhus.

35. Umans, Christopher; Villa, Tiziano; Sangiovanni-Vincentelli, Alberto Luigi (2006-06-05). "Complexity of two-level logic minimization". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 25 (7): 1230–1246. doi:10.1109/TCAD.2005.855944. S2CID 10187481.

36. Nelson, Victor P.; Nagle, H. Troy; Carroll, Bill D.; Irwin, J. David (1995). Digital Logic Circuit Analysis and Design (2 ed.). Prentice Hall. p. 234. ISBN 978-0-13463894-2. Retrieved 2014-08-26.

37. Feldman, Vitaly (2009). "Hardness of Approximate Two-Level Logic Minimization and PAC Learning with Membership Queries". Journal of Computer and System Sciences. 75: 13–25 [13–14].

38. Gimpel, James F. (1965). "A Method for Producing a Boolean Function Having an Arbitrary Prescribed Prime Implicant Table". IEEE Transactions on Computers. 14: 485–488. doi:10.1109/PGEC.1965.264175.

39. Paul, Wolfgang Jakob (1974). "Boolesche Minimalpolynome und Überdeckungsprobleme". Acta Informatica.

40. Зубчук В. И. и др. Довідник по цифровій схемотехніці / В. И. Зубчук, В. П. Сігорський, А. Н. Шкуро. — К. Тэхника, 1990. — 448 с.

41. Об’єктно-орієнтоване програмування [https://docs.microsoft.com/ru-ru/dotnet/csharp/fundamentals/object-oriented/]

42. Класи чи структури, в чому різниця [https://c-sharp.pro/%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D1%8B-%D0%B8%D0%BB%D0%B8-%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D1%8B-%D0%B2-%D1%87%D0%B5%D0%BC-%D0%BE%D1%82%D0%BB%D0%B8%D1%87%D0%B8%D1%8F/]

43. Типи значень та посилальні типи [https://metanit.com/sharp/tutorial/2.16.php]

44. Посилальні типи (довідник по C#) [https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/reference-types]

45. Пул об'єктів (шаблон проектування) [https://uk.wikipedia.org/wiki/%D0%9F%D1%83%D0%BB_%D0%BE%D0%B1%27%D1%94%D0%BA%D1%82%D1%96%D0%B2_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)]

46. Шаблони, що породжують: Пул об'єктів (Object pool) [https://andrey.moveax.ru/post/patterns-oop-creational-object-pool]

47. Object Pool Design Pattern [https://sourcemaking.com/design_patterns/object_pool/]

48. Kircher, Michael; Prashant Jain (2002-07-04). "Pooling Pattern". EuroPLoP 2002. Germany. Retrieved 2007-06-09.

49. Fundamentals of garbage collection [https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals]

50. Збирання сміття, управління пам'яттю та вказівники [https://metanit.com/sharp/tutorial/8.1.php]

51. Garbage Collection in C# | .NET Framework [https://www.geeksforgeeks.org/garbage-collection-in-c-sharp-dot-net-framework/]

Додаток А. ЛІСТИНГ ПРОГРАМИ

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;

/// <summary>
/// Базовий клас для логічних функцій
/// </summary>
public abstract class LogicFunction
{
  //Значення символа "склеєної" позиції '*'
  public const byte cStarSymb = 2;

  //Диз’юнкції або кон’юнкції функції
  public readonly ICollection<byte[]> Terms = new LinkedList<byte[]>();
  //Обчислення значення функції
  public abstract bool Calculate(bool[] X);
  // Обчислення значення функції
  public abstract bool Calculate(char[] X);
  // Обчислення значення функції
  public abstract bool Calculate(byte[] X);
}

/// <summary>
/// Диз’юнктивна нормальна форма
/// </summary>
public class Dnf : LogicFunction
{
  public override bool Calculate(byte[] X)
  {
```

```
            bool bResult = false;
            foreach (byte[] term in Terms)
            {
              bool bTermVal = true;
              for (int i = 0; i < term.Length; i++)
              {
                if ((term[i] >= cStarSymb) || (term[i] == X[i])) continue;
                bTermVal = false;
                break;
              }
              //bResult |= bTermVal;
              if (bTermVal)
              {
                bResult = true;
                break;
              }
            }
            return bResult;
          }

          public override bool Calculate(char[] X)
          {
            bool bResult = false;
            foreach (byte[] term in Terms)
            {
              bool bTermVal = true;
              for (int i = 0; i < term.Length; i++)
              {
                if ((term[i] >= cStarSymb) || (term[i] == (byte)(X[i] == '0' ? 0 : 1)))
continue;
                bTermVal = false;
```

```
      break;
    }
    //bResult |= bTermVal;
    if (bTermVal)
    {
      bResult = true;
      break;
    }
  }
  return bResult;
}

public override bool Calculate(bool[] X)
{
  bool bResult = false;
  foreach (byte[] term in Terms)
  {
    bool bTermVal = true;
    for (int i = 0; i < term.Length; i++)
    {
      if ((term[i] >= cStarSymb) || ((term[i] != 0) == X[i])) continue;
      bTermVal = false;
      break;
    }
    //bResult |= bTermVal;
    if (bTermVal)
    {
      bResult = true;
      break;
    }
  }
```

```
        return bResult;
      }
    }

    /// <summary>
    /// Мінімізація логічної функції методом Куайна - Мак-Класки
    /// </summary>
    public class Quine_McCluskey
    {
      //Максимальне кол-во елементів, при котрому використовується Dictionary,
      //щоб не відбулось переповнення контейнера
      private static readonly int DICT_MAX_ITEMS = 8000000;

      //Колекція "склеєних" термів
      private readonly Dnf _result = new Dnf();
      public Dnf Result
      {
        get { return _result; }
      }

      //Запуск метода
      public void Start(IEnumerable<byte[]> TermsInput)
      {
        LogicFuncMinimize(TermsInput, _result.Terms);
      }

      //Запуск метода
      public void Start(IEnumerable<char[]> TermsInput)
      {
        Start(TermsInput.Select(t => t.Select(p => (byte)(p == '0' ? 0 :
1)).ToArray()));
```

```
        }

        //Запуск метода
        public void Start(IEnumerable<bool[]> TermsInput)
        {
          Start(TermsInput.Select(t => t.Select(p => (byte)(p ? 1 : 0)).ToArray()));
        }

        //Запуск метода з читанням ДСНФ з файлу
        public void Start(string sFileName)
        {
          Start(LoadDsnfFromFile(sFileName));
        }

        //Знаходження мінімальної логічної функції
        private static void LogicFuncMinimize(
          IEnumerable<byte[]> InpTerms, ICollection<byte[]> OutTerms)
        {
          LinkedList<TreeNodeEnd> OutTemp = new LinkedList<TreeNodeEnd>();
          if (InpTerms.First().Length < 40)
          {
            IDictionary<UInt64, TreeNodeEnd> X1Tree = (InpTerms.Count() < DICT_MAX_ITEMS
?
              (IDictionary<UInt64, TreeNodeEnd>)(new Dictionary<UInt64, TreeNodeEnd>())
:
              new SortedDictionary<UInt64, TreeNodeEnd>());
            DeleteDublicatingTerms(InpTerms, X1Tree);
            // Повтор доти доки будуть залишатись терми
            while (X1Tree.Count != 0)
            {
              IDictionary<UInt64, TreeNodeEnd> X2Tree = (X1Tree.Count < DICT_MAX_ITEMS ?
```

```
                (IDictionary<UInt64, TreeNodeEnd>)(new Dictionary<UInt64,
TreeNodeEnd>()) :
                new SortedDictionary<UInt64, TreeNodeEnd>());
              Skleivanie(X1Tree, X2Tree, OutTemp);
              X1Tree.Clear();
              X1Tree = X2Tree;
              GC.Collect(); //Збирання сміття
            }
          }
          else
          {
            TreeFuncTerm X1Tree = new TreeFuncTerm();
            DeleteDublicatingTerms(InpTerms, X1Tree);
            //Повтор доти доки будуть залишатись терми
            while (X1Tree.Count != 0)
            {
              TreeFuncTerm X2Tree = new TreeFuncTerm();
              Skleivanie(X1Tree, X2Tree, OutTemp);
              X1Tree.Clear();
              X1Tree = X2Tree;
              GC.Collect(); // Збирання сміття
            }
          }
          ReduceRedundancyTerms(OutTemp, OutTerms);
        }

        //Процедура читання ДСНФ з файлу і запис в матрицю
        private static ICollection<byte[]> LoadDsnfFromFile(string sFullFileName)
        {
          ICollection<byte[]> DSNF = new LinkedList<byte[]>();
          //Чтение строк из файла
```

```
  using (StreamReader InFile = new StreamReader(sFullFileName))
  {
    string sLine = "";
    while ((sLine = InFile.ReadLine()) != null)
    {
      DSNF.Add(sLine.Select(p => (byte)(p == '0' ? 0 : 1)).ToArray());
    }
  }
  return DSNF;
}

//Видалення дублікатів термів з вхідного списку
//В вихідний словник додаються тільки унікальні терми
private static void DeleteDublicatingTerms(IEnumerable<byte[]> InX1,
  IDictionary<UInt64, TreeNodeEnd> OutX2Tree)
{
  OutX2Tree.Clear();
  foreach (byte[] x1 in InX1)
  {
    UInt64 iCode = GetTermCode(x1);
    if (OutX2Tree.ContainsKey(iCode)) continue;
    OutX2Tree.Add(iCode, new TreeNodeEnd(x1, null, null));
  }
}

//Видалення дублікатів термів з вхідного списку
//В вихідний словник додаються тільки унікальні терми
private static void DeleteDublicatingTerms(IEnumerable<byte[]> InX1,
  TreeFuncTerm OutX2Tree)
{
  OutX2Tree.Clear();
```

```
    foreach (byte[] x1 in InX1)
    {
      OutX2Tree.Add(x1, null, null);
    }
  }

  //Склеювання строк з однією різницею
  private static void Skleivanie(
    TreeFuncTerm X1Tree, TreeFuncTerm X2Tree,
    ICollection<TreeNodeEnd> OutResult)
  {
    Dictionary<int, TreeNodeEnd> FindTerms = new Dictionary<int, TreeNodeEnd>();
    TreeNodeEnd x1 = X1Tree.Last;
    while (x1 != null)
    {
      bool bIsSkleiv = false;
      for (int iPos = 0; iPos < x1.Term.Length; iPos++)
      {
        byte cSymbSav = x1.Term[iPos];
        if (cSymbSav == LogicFunction.cStarSymb) continue;
        // Склеювання двох термів з однією різницею
        x1.Term[iPos] = (byte)(1 - cSymbSav);
        TreeNodeEnd pSkleivNode = X1Tree.Contains(x1.Term);
        if (pSkleivNode != null)
        {
          bIsSkleiv = true;
          //Перевірка, щоб комбінації термів оброблялись тільки по одному разу
          if (cSymbSav == 1)
          {
            x1.Term[iPos] = LogicFunction.cStarSymb;
            X2Tree.Add(x1.Term, x1, pSkleivNode);
```

```
        }
      }
      x1.Term[iPos] = cSymbSav;
    }
    //Додавання на вихід термів, які ні з ким не склеїлись
    if (!bIsSkleiv) OutResult.Add(x1);
    //Перехід до наступного листа дерева
    x1 = x1.PrevNode;
  }
}

//Повертає унікальний код для терма
private static UInt64 GetTermCode(byte[] pTerm)
{
  UInt64 iMultip = 1, iCode = 0;
  for (int i = 0; i < pTerm.Length; i++)
  {
    iCode += (iMultip * pTerm[i]);
    iMultip *= 3;
  }
  return iCode;
}

//Склеювання строк з однією різницею
private static void Skleivanie(
  IDictionary<UInt64, TreeNodeEnd> X1Tree,
  IDictionary<UInt64, TreeNodeEnd> X2Tree,
  ICollection<TreeNodeEnd> OutResult)
{
  foreach (KeyValuePair<UInt64, TreeNodeEnd> x1 in X1Tree)
  {
```

```
            bool bIsSkleiv = false;
            UInt64 iMultip = 1;
            for (int iPos = 0; iPos < x1.Value.Term.Length; iPos++)
            {
              byte cSymbSav = x1.Value.Term[iPos];
              if (cSymbSav != LogicFunction.cStarSymb)
              {
                UInt64 iCode;
                if (cSymbSav == 0)
                  iCode = x1.Key + iMultip;
                else //_if (cSymbSav == 1)
                  iCode = x1.Key - iMultip;
                TreeNodeEnd pSkleivNode = null;
                X1Tree.TryGetValue(iCode, out pSkleivNode);
                if (pSkleivNode != null)
                {
                  bIsSkleiv = true;
                  //Перевірка, щоб комбінації термів оброблялись тільки по одному разу
                  if (cSymbSav == 1)
                  {
                    // Склеювання двох термів з однією різницею
                    iCode = x1.Key + iMultip;
                    if (!X2Tree.ContainsKey(iCode))
                    {
                      x1.Value.Term[iPos] = LogicFunction.cStarSymb; //Метка склеивания
                      X2Tree.Add(iCode, new TreeNodeEnd(x1.Value.Term, x1.Value,
pSkleivNode));
                      x1.Value.Term[iPos] = cSymbSav;
                    }
                  }
                }
```

```
        }
        iMultip *= 3;
      }
      //Додавання на вихід термів, які ні з ким не склеїлись
      if (!bIsSkleiv) OutResult.Add(x1.Value);
    }
  }

  //Відкидання терм за допомогою алгоритму приблизного рішення задачі о покритті
  private static void ReduceRedundancyTerms(
    IEnumerable<TreeNodeEnd> SkleivTerms, ICollection<byte[]> ResultTerms)
  {
    //Підготовка результуючого контейнера
    ResultTerms.Clear();
    //Контейнер для відповідності кінцевого терму до списку первинних термів, які
його образували
    IDictionary<TreeNodeEnd, HashSet<TreeNodeEnd>> Outputs2Inputs =
      new Dictionary<TreeNodeEnd, HashSet<TreeNodeEnd>>();
    //Контейнер для відповідності первинних вхідних терм до вихфдних, які їх
покривають
    IDictionary<TreeNodeEnd, HashSet<TreeNodeEnd>> Inputs2Outputs =
      new Dictionary<TreeNodeEnd, HashSet<TreeNodeEnd>>();
    //Збір статистики про покриття вихідними термами вхідних
    foreach (TreeNodeEnd outTerm in SkleivTerms)
    {
      //Контейнер вхідних термів, які покривають даний вихідний терм term
      HashSet<TreeNodeEnd> InpTermsLst = new HashSet<TreeNodeEnd>();
      HashSet<TreeNodeEnd> ListNumbers = new HashSet<TreeNodeEnd>();
      ListNumbers.Add(outTerm);
      while (ListNumbers.Count > 0)
      {
        TreeNodeEnd pCurrNode = ListNumbers.First();
```

```
          ListNumbers.Remove(pCurrNode);
          if (pCurrNode.Parent1 != null && pCurrNode.Parent2 != null)
          {
            ListNumbers.Add(pCurrNode.Parent1);
            ListNumbers.Add(pCurrNode.Parent2);
          }
          else
          {
            InpTermsLst.Add(pCurrNode);
            if (!Inputs2Outputs.ContainsKey(pCurrNode))
            {
              Inputs2Outputs.Add(pCurrNode, new HashSet<TreeNodeEnd>());
            }
            Inputs2Outputs[pCurrNode].Add(outTerm);
          }
        }
        Outputs2Inputs.Add(outTerm, InpTermsLst);
      }
      //Сортування словника вихідних термів за зростанням кіл-ті їх покриттів
вхідними
      Inputs2Outputs = Inputs2Outputs.OrderBy(p => p.Value.Count).ToDictionary(p =>
p.Key, v => v.Value);
      //Перебір всіх вхідних термів, посортованих за зростанням кіл-ті їх покриттів
вхідними

      while (Inputs2Outputs.Count > 0)
      {
        TreeNodeEnd outTerm = Inputs2Outputs.First().Value.OrderByDescending(q =>
Outputs2Inputs[q].Count()).First();
        ResultTerms.Add(outTerm.Term);
        foreach (TreeNodeEnd inpTerm in Outputs2Inputs[outTerm].ToArray())
        {
```

```
        foreach (TreeNodeEnd outTerm2Del in Inputs2Outputs[inpTerm])
        {
          Outputs2Inputs[outTerm2Del].Remove(inpTerm);
        }
        Inputs2Outputs.Remove(inpTerm);
      }
    }
  }
}

/// <summary>
/// Дерево термів
/// </summary>
public class TreeFuncTerm
{
  //Корінь
  private readonly TreeNodeMiddle rootNode = new TreeNodeMiddle();
  //Посилання на завершуючий вузол послідовності кінцевих листів дерева
  private TreeNodeEnd _lastTreeNode = null;
  public TreeNodeEnd Last
  {
    get { return _lastTreeNode; }
  }
  //Повертає кількість термів в дереві
  private int _count = 0;
  public int Count
  {
    get { return _count; }
  }

  //Конструктор
```

```
public TreeFuncTerm()
{
  Clear();
}

//Очистити дерево
public void Clear()
{
  rootNode.Clear();
  _count = 0;
  _lastTreeNode = null;
}

//Додавання в дерево нового терму
public void Add(byte[] term, TreeNodeEnd pParent1, TreeNodeEnd pParent2)
{
  TreeNodeMiddle pCurrNode = rootNode;
  int iTermLength1 = term.Length - 1;
  for (int i = 0; i < iTermLength1; i++)
  {
    byte cSymb = term[i];
    TreeNodeBase item = pCurrNode.Childs[cSymb];
    if (item == null)
    {
      item = new TreeNodeMiddle();
      pCurrNode.Childs[cSymb] = item;
    }
    pCurrNode = (TreeNodeMiddle)item;
  }
  TreeNodeBase pNewNode = pCurrNode.Childs[term[iTermLength1]];
  if (pNewNode == null)
```

```
    {
      pNewNode = new TreeNodeEnd(term, pParent1, pParent2, _lastTreeNode);
      pCurrNode.Childs[term[iTermLength1]] = pNewNode;
      _lastTreeNode = (TreeNodeEnd)pNewNode;
      _count++;
    }
  }

  //Перевірка знаходження послідовності в контейнері
  public TreeNodeEnd Contains(byte[] term)
  {
    TreeNodeBase pCurrNode = rootNode;
    foreach (byte cSymb in term)
    {
      pCurrNode = ((TreeNodeMiddle)pCurrNode).Childs[cSymb];
      if (pCurrNode == null) break;
    }
    return ((pCurrNode != null) && (pCurrNode is TreeNodeEnd) ?
      (TreeNodeEnd)pCurrNode : null);
  }
}

/// <summary>
/// Базовый интерфейс узла дерева термов
/// </summary>
public interface TreeNodeBase
{
  //Очистка выделенных ресурсов
  void Clear();
}
```

```
/// <summary>
/// Кінцевий вузол дерева термів
/// </summary>
public class TreeNodeEnd : TreeNodeBase
{
  //Терм, сопоставленный узлу
  public readonly byte[] Term;
  //Посилання на попередній кінцевий вузол дерева поточного рівня
  //для створення однозв’язного списку
  public readonly TreeNodeEnd PrevNode;
  //Посилання на батьківський кінцевий вузол дерева попереднього рівня
  public readonly TreeNodeEnd Parent1;
  //Посилання на батьківський кінцевий вузол дерева попереднього рівня
  public readonly TreeNodeEnd Parent2;

  //Конструктор
  public TreeNodeEnd(byte[] pTermRef, TreeNodeEnd pParent1, TreeNodeEnd pParent2,
    TreeNodeEnd pPrevTreeNode = null)
  {
    pTermRef.CopyTo(Term = new byte[pTermRef.Length], 0);
    Parent1 = pParent1;
    Parent2 = pParent2;
    PrevNode = pPrevTreeNode;
  }

  //Очистка ресурсів
  public void Clear() { }
}

/// <summary>
/// Проміжний вузол дерева термів
```

```
/// </summary>
public class TreeNodeMiddle : TreeNodeBase
{
  //Дочерні вузли
  public readonly TreeNodeBase[] Childs = new TreeNodeBase[3];

  //Очистка ресурсів
  public void Clear()
  {
    for (int i = 0; i < 3; i++)
    {
      if (Childs[i] != null)
      {
        Childs[i].Clear();
        Childs[i] = null;
      }
    }
  }
}

public class Programm
{
    public static void TestQuineMcCluskeyRandom(int iVariableAmount, string sFileName = "")
    {
        int iTotalCombines = 1 << iVariableAmount;
        ICollection<KeyValuePair<byte[], bool>> FuncResult = new List<KeyValuePair<byte[], bool>>(iTotalCombines);
        if (!String.IsNullOrEmpty(sFileName)) File.Delete(sFileName);
        Random rnd = new Random();
        //int iLogicFuncMask = 0;
        //while (iLogicFuncMask == 0) iLogicFuncMask = rnd.Next(iTotalCombines);
```

```
            for (int iCounter = 0; iCounter < iTotalCombines; iCounter++)
            {
                int iCurValue = iCounter;
                byte[] pTerm = new byte[iVariableAmount];
                for (int i = 0; i < iVariableAmount; i++)
                {
                    pTerm[i] = (byte)(iCurValue % 2);
                    iCurValue /= 2;
                }
                bool bFuncValue = (rnd.Next(2) != 0);
                //bool bFuncValue = (iCounter & iLogicFuncMask) > 0;
                if (bFuncValue && !String.IsNullOrEmpty(sFileName))
                {
                    File.AppendAllText(sFileName, pTerm.ToString() +
Environment.NewLine);
                }
                FuncResult.Add(new KeyValuePair<byte[], bool>(pTerm, bFuncValue));
            }
            //Положительные термы
            IEnumerable<byte[]> pTrueTerms = FuncResult.Where(p => p.Value).Select(p
=> p.Key);
            //Запуск в работу
            DateTime DtStart = DateTime.Now;
            Console.WriteLine("Старт - " + DtStart.ToLongTimeString());
            Quine_McCluskey Logic = new Quine_McCluskey();
            Logic.Start(pTrueTerms);
            DateTime DtEnd = DateTime.Now;
            TimeSpan Elapsed = DtEnd - DtStart;
            Console.WriteLine("Завершение - " + DtEnd.ToLongTimeString());
            Console.WriteLine("Длительность - " +
String.Format("{0:00}:{1:00}:{2:00}",
              Elapsed.Hours, Elapsed.Minutes, Elapsed.Seconds));
```

```
                //Порівняння результатів
                int iErrorsCounter = 0;
                foreach (KeyValuePair<byte[], bool> kvp in FuncResult)
                {
                    if (Logic.Result.Calculate(kvp.Key) != kvp.Value) iErrorsCounter++;
                }
                Console.WriteLine("Кол-во входных переменных = " + iVariableAmount);
                Console.WriteLine("Кол-во входных термов = " + iTotalCombines);
                Console.WriteLine("Кол-во дизъюнктивных термов = " +
Logic.Result.Terms.Count);
                Console.WriteLine("Кол-во ошибок = " + iErrorsCounter);
                Console.ReadLine();
            }
        }
```

Додаток Б.

ВІДГУК КЕРІВНИКА

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ «ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника ____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

на магістерську роботу

студента ____
(прізвище, ім'я, по батькові)

курсу II групи ______________
спеціальності ____
освітньої програми ____
на тему ____
____

Актуальність теми ____
____
____
____

Мета досліджень ____
____
____
____

Коротка характеристика розділів роботи ______________________________
____
____
____
____

____

____

____

Практичне значення роботи ____

____

____

____

____

Зауваження та недоліки ____

____

____

____

____

Висновки та оцінка ____

____

____

____

Науковий
керівник ____
(прізвище, ім’я, по батькові, посада, місце роботи)

«___»______________20__ р. ______________________
(підпис)

Додаток В. РЕЦЕНЗІЯ

РЕЦЕНЗІЯ
на магістерську роботу

студента ____
(прізвище, ім'я, по батькові)

курсу II групи ____________
кафедри програмного забезпечення комп'ютерних систем
спеціальності ____
освітньої програми ____________________________________
Тема роботи ____

Стисла характеристика розділів роботи ____________________
____
____
____
____
____

Пропозиції, внесені студентом, рівень їх наукового обґрунтування __________
____
____
____
____

Практичне значення роботи ____________________________
____
____

Якість оформлення роботи _____________________________
____

Недоліки в роботі ____________________________________
____
____

Загальний висновок ___________________________________
(підготовленість студента до самостійної роботи як спеціаліста)

Оцінка магістерської роботи ___________________________

Рецензент ____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«___»____________20__ р. ____________________
(підпис)

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім'я файлу	**Опис**
Пояснювальні документи	
Диплом_Рибка.doc	Пояснювальна записка до магістерської роботи. Документ Word.
Диплом_Рибка.pdf	Пояснювальна записка до магістерської роботи в форматі PDF.
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму.
Презентація	
Презентація_Рибка.ppt	Презентація до магістерської роботи