Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Факультет комп’ютерних наук

**КУРСОВА РОБОТА**

з дисципліни «Теорія алгоритмів»

Тема: Задача про хід коня

Виконав студент 2 курсу
Групи КС-21
Клочко Андрій Володимирович
Перевірив:
доц. Щебенюк В.С .....

Харків – 2020

**ЗМІСТ**

# ВСТУП

Дуже велика кількість математичних задач і головоломок виникає при появі на дошці шахових фігур. Серед завдань, пов'язаних з їх маршрутами, найзнаменитішою є задача про хід коня.

Вона формулюється наступним чином: «Обійти конем всі клітинки шахівниці, займаючи кожне з них рівно один раз»

Особлива популярність завдання пояснюється тим, що у XVIII і XIX століттях нею займалися багато математиків, в тому числі великий Леонард Ейлер, який присвятив їй мемуари "Розв’язання одного цікавого питання, який, здається, не підпорядковується жодному дослідженню". Хоча задача була відома і до Ейлера, лише він вперше звернув увагу на її математичну сутність, і тому завдання часто пов'язують з його ім'ям [1].

# РОЗДІЛ 1 ОБЄКТ, ПРЕДМЕТ, МЕТА

**Об’єкт** – Задача про хід коня.

**Предмет** – Методи вирішення задачі про хід коня. Алгоритмічні методи вирішення задачі про хід коня.

**Мета** – Дослідити методи вирішення задачі про хід коня. Порівняти між собою методи вирішення Ейлера, Вандермонда, Варнсдорфа та програмно реалізувати і порівняти алгоритм пошуку з поверненням та алгоритм за методом Варнсдорфа.

# РОЗДІЛ 2 ХІД КОНЯ

## 2.1 Для тих, хто не знайомий з шахами

Для людини, яка не знайома з шахами, кінь ходить два квадрати горизонтально та один квадрат вертикально, або два квадрати вертикально та один квадрат горизонтально. В простолюдді, кінь ходить «буквою Г» (англ. версія «L»). Приклад ходу коня показано на рисунку 1.1.

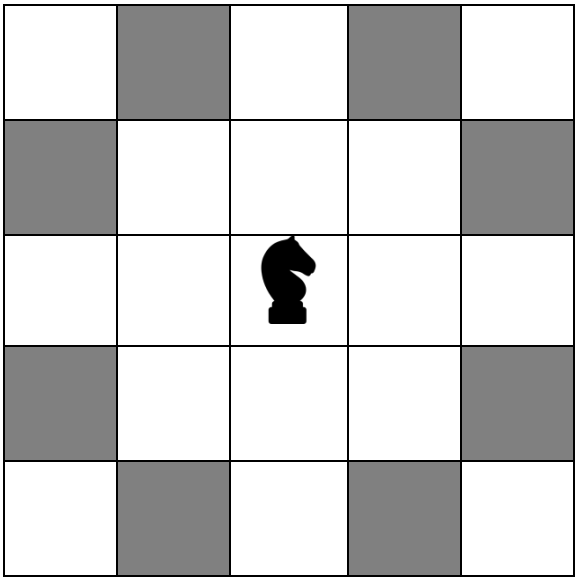

Рисунок 2.1 – Приклад ходу коня в шахах

## 2.2 Зв’язок задачі про хід коня з теорією графів

З точки зору теорії графів завдання про хід коня є окремим випадком важливої проблеми – знаходження Гамільтонового шляху у графі, тобто шляху, що проходить через всі його вершини по одному разу. Цим і пояснюється популярність завдання про хід коня в літературі з теорії графів, при цьому розглядається «граф коня» (рис. 1.2).

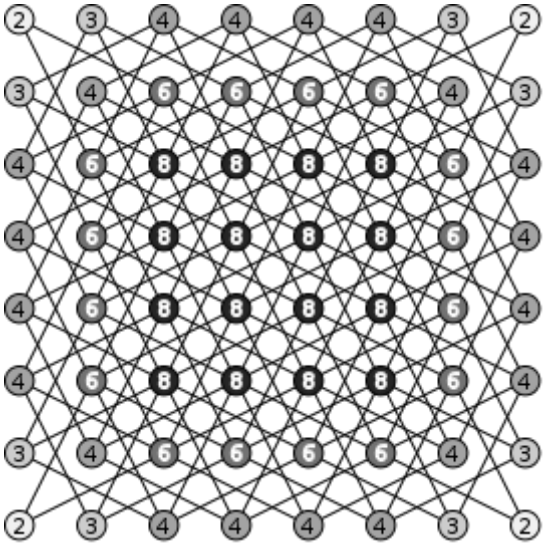

Рисунок 2.2 – Граф зв’язків між клітинками шахівниці

Завдання знаходження Гамільтонового шляху в графі є, у свою чергу, окремим випадком так званої задачі комівояжера, до якої зводяться найрізноманітніші завдання одного з найважливіших розділів прикладної математики – дослідження операцій. Потрібно знайти найкоротший шлях комівояжера, за яким він повинен об'їхати ряд міст (пов'язаних між собою деяким числом доріг), відвідавши кожен з них по одному разу. Звичайно, перш за все тут виникає питання, чи може комівояжер взагалі об'їхати всі міста з одноразовим відвідуванням кожного з них. Таким чином, можна вважати, що курсова робота присвячена подорожам по шахівниці «коня-комівояжера» [2].

### 2.3 Види маршрутів ходу коня

Бувають два види маршрутів ходу коня:

1) Замкнуті
2) Незамкнуті

При замкнутому проході коня потрібно відвідати всі поля шахівниці, після чого повернутися в початкове поле. Замкнуті маршрути існують на дошках $N \times N$ для всіх парних сторін дошки $N \geq 6$.

Незамкнутий варіант відрізняється від замкнутого тим, що в ньому не потрібно повертатися в початкову позицію. Незамкнуті маршрути існують на квадратних дошках $N \times N$ для всіх $N \geq 5$.

# РОЗДІЛ 3 МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ ПРО ХІД КОНЯ

## 3.1 Основні методи вирішення задачі про хід коня

Існує доволі багато різних методів вирішення цієї задачі, але в своїй курсовій роботі я хотів би зупинитися на трьох основних методах, а саме:

1) Метод Ейлера
2) Метод Вандермонда
3) Метод Варнсдорфа

## 3.2 Метод Ейлера

Ейлер починав з випадкового переміщення коня над дошкою, поки доступних ходів більше не ставало. Останні клітинки, які не потрапили під хід коня він помічав їх як a,b,…. Його метод полягав у встановленні певних правил, за якими ці мічені клітини можуть бути вставлені на хід коня, і також правила для повторного введення рішення. Також метод Ейлера дозволяє зробити з незамкнутого шляху коня – замкнутий за допомогою деяких перетворень, які ми розглянемо трішки пізніше [3].

Візьмемо приклад шляху, утвореного конем, з чотирма клітинками, що залишилися порожніми. Позначимо ці клітинка як a, b, c, d (рис. 3.1).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 55 | 58 | 29 | 40 | 27 | 44 | 19 | 22 |
| 60 | 39 | 56 | 43 | 30 | 21 | 26 | 45 |
| 57 | 54 | 59 | 28 | 41 | 18 | 23 | 20 |
| 38 | 51 | 42 | 31 | 8 | 25 | 46 | 17 |
| 53 | 32 | 37 | *a* | 47 | 16 | 9 | 24 |
| 50 | 3 | 52 | 33 | 36 | 7 | 12 | 15 |
| 1 | 34 | 5 | 48 | *b* | 14 | *c* | 10 |
| 4 | 49 | 2 | 35 | 6 | 11 | *d* | 13 |

Рисунок 3.1 – Незавершений шлях коня

Нам потрібно переробити незавершений шлях від 1 до 60 в завершений замкнутий тур коня.

З клітинки 1 можна піти в клітинку p, де p дорівнює 32, 52 або 2. З клітинки 60 можна піти в клітинку q, де q дорівнює 29, 59 або 51.

Якщо будь-яке зі значення p та q відрізняється на 1, ми можемо переробити шлях.

В нашому випадку p=52, q=51. Тому числа від 52 до 60 включно записуємо на дошці в оберненому порядку (рис 3.2). Як результат, маємо незавершений замкнутий шлях від 1 до 60.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 57 | 54 | 29 | 40 | 27 | 44 | 19 | 22 |
| 52 | 39 | 56 | 43 | 30 | 21 | 26 | 45 |
| 55 | 58 | 53 | 28 | 41 | 18 | 23 | 20 |
| 38 | 51 | 42 | 31 | 8 | 25 | 46 | 17 |
| 59 | 32 | 37 | *a* | 47 | 16 | 9 | 24 |
| 50 | 3 | 60 | 33 | 36 | 7 | 12 | 15 |
| 1 | 34 | 5 | 48 | *b* | 14 | *c* | 10 |
| 4 | 49 | 2 | 35 | 6 | 11 | *d* | 13 |

Рисунок 3.2 – Перетворення маршруту в незавершений замкнутий

Наступним кроком потрібно додати клітинки a, b, c, d до нашого маршруту.

В новому шляху, клітинка 60 посилається на клітинки 51, 53, 41, 25, 7, 5, або 3.

Не важливо яку з цих клітино ми візьмемо, але краще взяти клітинку 51, щоб вже з неї продовжити шлях в a, b та d.

Щоб це зробити, нам потрібно збільшити кожне число в клітинці на різницю останньої клітинки з обраною нами, отже 60 - 51 = 9 (рис. 3.3).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 66 | 63 | 38 | 49 | 36 | 53 | 28 | 31 |
| 61 | 48 | 65 | 52 | 39 | 30 | 35 | 54 |
| 64 | 67 | 62 | 37 | 50 | 27 | 32 | 29 |
| 47 | 60 | 51 | 40 | 17 | 34 | 55 | 26 |
| 68 | 41 | 46 | *a* | 56 | 25 | 18 | 33 |
| 59 | 12 | 69 | 42 | 45 | 16 | 21 | 24 |
| 10 | 43 | 14 | 57 | *b* | 23 | *c* | 19 |
| 13 | 58 | 11 | 44 | 15 | 20 | *d* | 22 |

Рисунок 3.3 – Збільшенні на 9 всі числа

Тепер заміняємо всі числа від 61 до 69 на числа від 1 до 9 включно, що в результаті дасть нам шлях від 1 до 60, в якому ми можемо продовжити свій хід в клітинки a, b, d (рис 3.4).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 38 | 49 | 36 | 53 | 28 | 31 |
| 1 | 48 | 5 | 52 | 39 | 30 | 35 | 54 |
| 4 | 7 | 2 | 37 | 50 | 27 | 32 | 29 |
| 47 | 60 | 51 | 40 | 17 | 34 | 55 | 26 |
| 8 | 41 | 46 | 61 | 56 | 25 | 18 | 33 |
| 59 | 12 | 9 | 42 | 45 | 16 | 21 | 24 |
| 10 | 43 | 14 | 57 | 62 | 23 | *c* | 19 |
| 13 | 58 | 11 | 44 | 15 | 20 | 63 | 22 |

Рисунок 3.4 – Залучення клітинок a, b, d до шляху коня

Нам залишається залучити клітинку c до нашого шляху.

Клітинка c посилається на клітинку 25, а клітинка 63 на 24. Ми можемо використати метод, яким користувались раніше щоб знову переробити шлях від 63 до 25 задом наперед. В результаті ми отримуємо хід до клітинки c та повний незамкнутий маршрут коня (рис. 3.5).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 50 | 39 | 52 | 35 | 60 | 57 |
| 1 | 40 | 5 | 36 | 49 | 58 | 53 | 34 |
| 4 | 7 | 2 | 51 | 38 | 61 | 56 | 59 |
| 41 | 28 | 37 | 48 | 17 | 54 | 33 | 62 |
| 8 | 47 | 42 | 27 | 32 | 63 | 18 | 55 |
| 29 | 12 | 9 | 46 | 43 | 16 | 21 | 24 |
| 10 | 45 | 14 | 31 | 26 | 23 | 64 | 19 |
| 13 | 30 | 11 | 44 | 15 | 20 | 25 | 22 |

Рисунок 3.5 – Повний незамкнутий маршрут коня

Як вже раніше згадувалося, метод Ейлера дозволяє зробити замкнутий шлях з незамкнутого. Для прикладу візьмемо незамкнутий маршрут коня на рисунку 3.5.

Нам потрібно зробити клітинку 64 ближче до клітинки 1. Зробимо це за допомогою клітинки 28 яка посилається на клітинку 1 та 27.

Запишемо шлях від 1 до 27 задом наперед (рис. 3.6).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 22 | 25 | 50 | 39 | 52 | 35 | 60 | 57 |
| 27 | 40 | 23 | 36 | 49 | 58 | 53 | 34 |
| 24 | 21 | 26 | 51 | 38 | 61 | 56 | 59 |
| 41 | 28 | 37 | 48 | 11 | 54 | 33 | 62 |
| 20 | 47 | 42 | 1 | 32 | 63 | 10 | 55 |
| 29 | 16 | 19 | 46 | 43 | 12 | 7 | 4 |
| 18 | 45 | 14 | 31 | 2 | 5 | 64 | 9 |
| 15 | 30 | 17 | 44 | 13 | 8 | 3 | 6 |

Рисунок 3.6 – Шлях від 1 до 27 задом наперед

З клітинки 1 можна піти в клітинки 26, 38, 54, 12, 2, 14, 16, 28. З клітинки 64 можна піти в клітинки 13, 43, 64, 55.

Клітинки 13 та 14 підходять нам через те, що у них різниця дорівнює 1. Отже записуємо хід від клітинки 1 до клітинки 13 задом наперед та отримуємо замкнутий шлях (рис. 3.7) [4].

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 22 | 25 | 50 | 39 | 52 | 35 | 60 | 57 |
| 27 | 40 | 23 | 36 | 49 | 58 | 53 | 34 |
| 24 | 21 | 26 | 51 | 38 | 61 | 56 | 59 |
| 41 | 28 | 37 | 48 | 3 | 54 | 33 | 62 |
| 20 | 47 | 42 | 13 | 32 | 63 | 4 | 55 |
| 29 | 16 | 19 | 46 | 43 | 2 | 7 | 10 |
| 18 | 45 | 14 | 31 | 12 | 9 | 64 | 5 |
| 15 | 30 | 17 | 44 | 1 | 6 | 11 | 8 |

Рисунок 3.7 – Повний замкнутий шлях коня

### 3.3 Метод Вандермонда

Другим автором після Ейлера був Вандермонд. Вандермонд спробував звести задачу до арифметичної.

Для цього він позначав маршрут коня по дошці у вигляді послідовності дробів x / y, де x і y – координати поля на дошці (рис. 3.8). Можна побачити, що в послідовності дробів різниця чисельників двох сусідніх дробів може бути тільки 1 або 2, відповідно, різниця знаменників сусідніх дробів становить також 1 або 2. Крім того, чисельник і знаменник не можуть бути менше 1 і більше 8.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 39 | 42 | 37 | 34 | 63 | 8 | 13 | 10 |
| 2 | 36 | 33 | 40 | 43 | 14 | 11 | 62 | 7 |
| 3 | 41 | 38 | 35 | 2 | 5 | 64 | 9 | 12 |
| 4 | 32 | 3 | 44 | 49 | 20 | 15 | 6 | 61 |
| 5 | 45 | 22 | 31 | 4 | 1 | 60 | 51 | 16 |
| 6 | 28 | 25 | 48 | 21 | 50 | 19 | 54 | 57 |
| 7 | 23 | 46 | 27 | 30 | 59 | 56 | 17 | 52 |
| 8 | 26 | 29 | 24 | 47 | 18 | 53 | 58 | 55 |

**5/5, 4/3, 2/4, 4/5, 5/3, 7/4, 8/2, 6/1, 7/3, 8/1, 6/2, 8/3, 7/1, 5/2, 6/4, 8/5, 7/7, 5/8, 6/6, 5/4, 4/6, 2/5, 1/7, 3/8, 2/6, 1/8, 3/7, 1/6, 2/8, 4/7, 3/5, 1/4, 2/2, 4/1, 3/3, 1/2, 3/1, 2/3, 1/1, 3/2, 1/3, 2/1, 4/2, 3/4, 1/5, 2/7, 4/8, 3/6, 4/4, 5/6, 7/5, 8/7, 6/8, 7/6, 8/8, 6/7, 8/6, 7/8, 5/7, 6/5, 8/4, 7/2, 5/1, 6/3**

Рисунок 3.8 – Приклад представлення маршруту коня методом Вандермонда

Хоч його метод розвинув досить велику популярність, сам Вандермонд зробив лише один тур на дошці 8 × 8, та описав це в статті «Проблемні ситуації».

Він починає з того, що покриває дошку чотирма подібними схемами, які разом утворюють шаблон. Він з'єднує протилежні пари цих ланцюгів, видаляючи пару паралельних ходів і приєднуючи вільні кінці, щоб отримати дві однакові схеми, кожна з яких має діаметральну симетрію, які разом утворюють псевдотур. Нарешті він з'єднує ці два ланцюги одним і тим же методом, щоб сформувати справжній тур, який, однак, не зберігає симетрію (рис. 2.9).

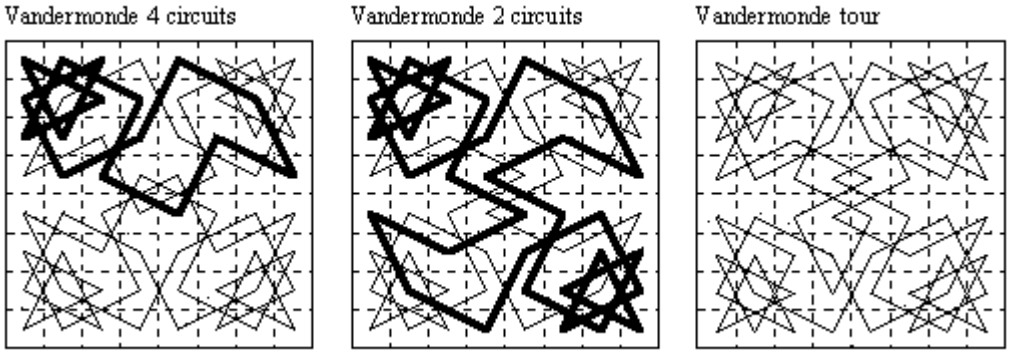


Рисунок 3.9 – Кінний тур зроблений Вандермондом

Через особливість створення кінного туру за допомогою симетрії, метод Вандермонда працює тільки на дошках в яких висота та ширина – парні значення.

### 3.4 Метод Варнсдорфа

Правило Варнсдорфа формулюється так: наступний хід коня потрібно робити на клітинку, звідки існує найменша кількість можливих ходів. Якщо клітинок з однаковою кількістю ходом декілька, то можна вибрати будь-яку.

На практиці це реалізується, наприклад, наступним чином. Перед кожним ходом коня визначається «рейтинг» найближчих доступних полів, на яких кінь ще не побував, і на які він може перейти за один хід. Рейтинг поля визначається числом найближчих доступних з нього полів. Чим менше рейтинг, тим він краще. Потім робиться хід на поле з найменшим рейтингом, якщо таких ходів декілька, можна піти в будь-яке.

Цей алгоритм відноситься до класу жадібних алгоритмів з евристичним методом вирішення через те, що хоч метод вирішення і є евристичним, але

відштовхуючись від цієї евристики ми підбираємо найкращій для нас хід, тобто виконується правило жадібних алгоритмів.

Евристика завжди працює на дошках від 5x5 до 76x76 клітинок, при більших розмірах дошки, може зайти в глухий кут. Крім того, базуючись на правилах алгоритму, не дає всіх можливих рішень (тобто ходів коня) [5].

# РОЗДІЛ 4 РЕЗУЛЬТАТИ РОБОТИ

## 4.1 Методи оптимізації

Так як в коня є потенційно 8 можливих ходів, що можна побачити на рисунку 1.1, створюємо два масиви та заповнюємо його можливими ходами (рис. 4.1).

```
const int possibleKnightMoveX[] = { -1, -1, 1, 1, -2, -2, 2, 2 };
const int possibleknightMoveY[] = { 2, -2, 2, -2, 1, -1, 1, -1 };
```

Рисунок 4.1 – Визначення всіх можливих ходів коня

За допомогою такого трюку, ми можемо не створювати окрему функцію з восьма блоками if else. Це дасть доволі гарний виграш в випадку алгоритму пошуку з поверненням так як нам не потрібно буде кожного разу перевіряти те, що ми уже перевірили.

Одним з методів оптимізації також є визначення клітинок з яких кінний тур буде неможливий. Якщо кількість клітинок дошки непарне, то туру з деяких клітинок не існує.

Для пояснення цього методу оптимізації, треба уточнити, що кінний тур проходить через клітинки, які чергуються по кольору.

Якщо загальна кількість клітинок непарна, то перша та остання клітинка шляху, пройденого конем, будуть одного і того ж кольору. Таким чином, шлях буде існувати тільки тоді, коли він починається з клітинки того кольору, який має найбільшу кількість клітинок (лістинг А.1) [6].

## 4.1 Алгоритм пошуку з поверненням

Створимо двомірний масив розміром ширини та висоти шахівниці, наприклад користувач введе висоту 8 та ширину 8, отже розмір масиву буде відповідно 8 × 8. Цей масив нам потрібен для перевірки в яких клітинках ми вже побували.  Якщо ми ще не були у відповідній клітинці, то елемент масиву буде дорівнювати 0, а якщо були – число від 1 до кількості всіх клітинок дошки.

Рекурсивна функція пошуку всіх можливих ходів робить це тільки для однієї клітинки, тому для того щоб розрахувати всі можливі варіанти треба зробити виклик рекурсивної функції в циклі для кожної клітинки окремо.

При вході в функцію пошуку, першим чином йде перевірка на те, чи всі клітинки шахівниці ми пройшли. Зробити це досить легко, для цього можна звіряти кількість пройдених елементів з максимальною кількістю всіх елементів (ширина×висота). Якщо виконається така перевірка, ми збільшуємо лічильник кількості турів коня, зменшуємо лічильник пройдених клітинок, анулюємо елемент масиву та повертаємося назад до попереднього стану кінного туру.

Якщо перевірка на те, що ми побували у всіх клітинках не виконується, перевіряємо в циклі від 0 до 8 в якому ми будемо перебирати всі можливі варіанти на хід коня. Якщо ми можемо зробити хід в наступну клітинку, викликаємо цю функцію.

Програма може знаходити шляхи для дошок не однакової ширини та висоти.

Перевіримо правильність роботи алгоритму.

Введемо розмір дошки 5 на 5 (рис. 4.2).

```
Enter the width of desk: 5
Enter the heigth of desk: 5
Do you want to print knight tour? (0 - No, 1 - Yes): 0
Calculating...
1728 Tours
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Рисунок 4.2 – Кількість турів на дошці 5 на 5

Для перевірки правильності розрахунку, можна звіритись з онлайн-енциклопедією цілочисельних послідовностей (рис. 4.3) [7].

```
# A165134 (b-file synthesized from sequence entry)
1 1
2 0
3 0
4 0
5 1728
6 6637920
7 165575218320
8 19591828170979904
```

Рисунок 4.3 – Результати розрахунків з онлайн ресурсу OEIS

Також можна подивитися обмежену кількість турів та розрахунок туру на нерівних дошках (рис. 3.4).

```
Enter the width of desk: 12
Enter the heigth of desk: 3
Do you want to print knight tour? (0 - No, 1 - Yes): 1
How much knight tour's you want to print?
To print ALL enter 'YOUR NUMBER' < 1
Enter the number: 3
1       20      3       6       17      8       33      10      13      26      31      28
36      5       18      21      34      11      16      23      32      29      14      25
19      2       35      4       7       22      9       12      15      24      27      30


1       20      3       6       17      8       33      10      13      24      27      30
36      5       18      21      34      11      16      23      32      29      14      25
19      2       35      4       7       22      9       12      15      26      31      28


1       22      3       6       25      8       33      30      27      18      15      12
36      5       24      21      34      31      26      19      10      13      28      17
23      2       35      4       7       20      9       32      29      16      11      14
```

Рисунок 4.4 – Приклад туру для нерівного розміру дошки

Хоча за допомогою такого методу можна точно перевірити кількість всіх можливих турів, час виконання такого алгоритму вже навіть для дошки 6 на 6 виконується дуже довго через те, що складність такого алгоритму складає $O(8^{n^2})$.

На алгоритмі пошуку з поверненням можна оцінити результативність оптимізації за допомогою знаходження клітинок з яких немає ходу, для прикладу візьмемо дошку розміром 5 × 5 (рис. 4.5).

```
Time without optimization: 4640
Time with optimization: 2365
```

4.5 – Демонстрація ефективності методу оптимізації

Повний вихідний код програми можна подивитися в лістингу А.2.

### 4.2 Алгоритм за методом Варнсдорфа

Також створюємо 2 масиви можливих ходів для коня (рис. 4.1).

При вході в функцію пошуку туру нас зустрічає цикл від 0 до кількості всіх клітинок на дошці (висота × ширина), в якому є ще один вкладений цикл від 0 до 8 (кількість можливих ходів коня).

Другий вкладений цикл створений для того, щоб перевірити всі можливі варіанти для ходу та порівняти, який з цих варіантів нам найбільше підходить, а якщо варіантів більше немає, спеціально створеній для таких

ситуацій змінній буде присвоєно значення -1, що дасть змогу зрозуміти, коли можливі варіанти для ходу скінчилися.

По закінченню цього циклу виконується перевірка на те, чи скінчилися в нас варіанти для ходу.

Після успішного проходження всіх перевірок, змінюється поточна позиція коня на те місце, яке ми розрахували раніше в циклі.

Подивимося на результати для дошки 8 на 8 (рис. 4.6).

```
Enter the height of chessboard: 8
Enter the width of chessboard: 8


Starting from [0][0]
Number of passed cells: 64
1       48      15      32      57      28      13      30
16      33      60      53      14      31      64      27
49      2       47      56      61      58      29      12
34      17      54      59      52      45      26      63
3       50      43      46      55      62      11      40
18      35      20      51      44      41      8       25
21      4       37      42      23      6       39      10
36      19      22      5       38      9       24      7
```

Рисунок 4.6 – Метод Варнсдорфа для дошки 8 на 8 з клітинки 1/1

Як можна спостерігати, евристика гарно спрацювала на таких розмірах дошки, але як вже було згадано раніше в теоретичній частині, шанс не отримати кінного туру на великих розмірах набагато більше.

Тепер подивимося на результати розрахунку для дошки 100 на 100 (рис. 4.7).

```
Starting from [0][0]
Number of passed cells: 9950

Starting from [0][1]
Number of passed cells: 9998

Starting from [0][2]
Number of passed cells: 9956

Starting from [0][3]
Number of passed cells: 9999
```

Рисунок 4.7 – Метод Варнсдорфа для дошки 100 на 100

Тепер же можна спостерігати зовсім іншу картину. Шанс того, що такий метод дасть вірний тур уже не дуже високий, а то і зовсім низький. Найближчий вірний кінний тур методом Варнсдорфа на дошці 100 × 100 буде здійснений в позиції 11/1 (рис. 4.8).

```
Starting from [0][10]
Number of passed cells: 10000
```

Рисунок 4.8 – Вірний кінний тур для дошки 100 × 100

Як і було сказано раніше, такий метод не дає нам всіх обходів, але через те, що складність алгоритм виконується за час $O(n)$, розрахунки проводяться набагато швидше порівняно з алгоритмом пошуку з поверненням.

Повний вихідний код можна подивитися в лістингу А.3.

### 4.3 Евристична евристика

При написанні алгоритму знаходження кінного туру методом Варнсдорфа, мені стало цікаво, а як буде працювати алгоритм, якщо робити хід не в клітинку з найменшою кількістю можливих ходів, а навпаки, коли ми будемо вибирати клітинку з найбільшою кількістю.

Були перевірені всі варіанти для дошок від 5 × 5 до 10 × 10 разом з дошками з нерівними сторонами, але не було знайдено жодного результату при якому кількість пройдених клітинок дорівнювала би кількості всіх клітинок на дошці. Майже в кожному випадку було пройдено всього половина шляху.

Хоч ці результати нам нічого і не дали, та все ж було цікаво це перевірити.

# ВИСНОВОК

В своїй курсовій роботі я теоретично розібрав три методи та програмно реалізував два методи вирішення задачі про хід коня.

Так як алгоритми пошуку з поверненням та Варнсдорфа категорично відрізняються за методами знаходження шляху коня, а також через те, що пошук з поверненням на відміну від алгоритму Варнсдорфа знаходить всі можливі шляхи коня, що в рази збільшує кількість розрахунків, порівнювати ці два методи за часовими показниками майже немає сенсу.

Також через категоричні відмінності в результатах роботи цих двох алгоритмів, важко сказати, який з них краще в якому випадку, а який ні, але судячи з того, що звичайному користувачу зазвичай не потрібні всі кінні тури, алгоритм Варнсдорфа в такому випадку буде набагато кращий, через свою швидкість роботи (якщо хід коня можна знайти цим алгоритмом).

Задачу про хід коня часто зрівняють з задачею комівояжера. Так як задача комівояжера відноситься до реального життя набагато більше ніж задача про хід коня (пошук найкоротшого шляху на карті, різні види спорту, наприклад орієнтування), затребуваність на оптимізацію вирішенні задачі про хід коня не дуже висока.

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ


1) Knight’s tour // wikipedia. URL: https://en.wikipedia.org/wiki/Knight%27s_tour. Дата звернення: 13.11.2020.
2) Задача о ходе коня. Связь задачи о ходе коня с теорией графов. URL: https://forany.xyz/a-16?pg=8. Дата звернення: 13.11.2020.
3) Rediscovery of the Knight's Problem. Euler 1759. URL: https://www.mayhematics.com/t/1b.htm. Дата звернення: 19.11.2020.
4) Colleen Raimondi. Презентація: The Knight’s Tour. Euler’s Method. URL: https://goo-gl.su/sjoAH. Дата звернення: 19.11.2020
5) Обход доски шахматным конём. Правило Варнсдорфа. URL: http://algolist.manual.ru/maths/combinat/knight.php. Дата звернення: 13.11.2020.
6) Методы оптимизации. Определение клеток, обход из которых невозможен. URL: http://is.ifmo.ru/download/knight.pdf. Дата звернення: 08.12.2020.
7) Online Encyclopedia of Integer Sequences // OEIS. URL: https://oeis.org/A165134/b165134.txt. Дата звернення: 08.12.2020.

# ДОДАТОК А. ВИХІДНИЙ КОД

## Лістинг А.1 – Функція перевірки неможливості туру з клітинки

```
int tourIsPossible(int posInRow, int posInColumn){
     if(((chessBoardWidth * chessBoardHeight)%2 != 0)
          && ((posInRow + posInColumn)%2 != 0)){
          return 0;
     }
     return 1;
}
```

## Лістинг А.2 – Вихідний код програми пошуку з поверненням

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>

//functions
void countAllKnightMoveTours();
int tourIsPossible(int posInRow, int posInColumn);
int** initTwoDimentionalArray(int columns, int rows);
void countKnightMovesFromOneCell(int posInRow, int posInColumn, int
**chessBoard);
void printChessBoard(int **chessBoard);

//variables
int chessBoardWidth;
int chessBoardHeight;
int knightTourCounter = 0; //current number of passed cells
int numberOfKnightTours = 0; //summary number of knight tours
int maxCountOfKnightTour;
int printable;
const int possibleKnightMoveX[] = { -1, -1, 1, 1, -2, -2, 2, 2 };
const int possibleknightMoveY[] = { 2, -2, 2, -2, 1, -1, 1, -1 };

int main(int argc, char *argv[]) {

     printf("Enter the width of desk: ");
     scanf("%d", &chessBoardWidth);

     printf("Enter the heigth of desk: ");
     scanf("%d", &chessBoardHeight);


     printf("Do you want to print knight tour? (0 - No, 1 - Yes): ");
     scanf("%d", &printable);

     if(printable == 1){
          printf("How much knight tour's you want to print?\nTo print
ALL enter 'YOUR NUMBER' < 1\nEnter the number: ");
          scanf("%d", &maxCountOfKnightTour);

          countAllKnightMoveTours();
```

```
	}else if(printable == 0){
		maxCountOfKnightTour = 0;

		printf("Calculating...\n");
		countAllKnightMoveTours();
		printf("%d Tours", numberOfKnightTours);
	}

	return 0;
}

void countAllKnightMoveTours(){

	int **chessBoard = initTwoDimentionalArray(chessBoardWidth,
chessBoardHeight);

	int i,j;
	for(i=0 ; i<chessBoardHeight ; i++){
		for(j=0 ; j<chessBoardWidth ; j++){
			if(tourIsPossible(i, j)){
				countKnightMovesFromOneCell(i, j, chessBoard);
			}
		}
	}
}

int tourIsPossible(int posInRow, int posInColumn){
	if(((chessBoardWidth * chessBoardHeight)%2 != 0)
		&& ((posInRow + posInColumn)%2 != 0)){
		return 0;
	}
	return 1;
}

int** initTwoDimentionalArray(int columns, int rows){

	int **array = (int**)calloc(rows, sizeof(int*));

	int i;
	for(i=0 ; i<rows ; i++){
		array[i] = (int*)calloc(columns, sizeof(int));
	}

	return array;
}

void countKnightMovesFromOneCell(int posInRow, int posInColumn, int
**chessBoard){

	knightTourCounter++;
	chessBoard[posInRow][posInColumn] = knightTourCounter;

	if(knightTourCounter == chessBoardWidth*chessBoardHeight){
		numberOfKnightTours++;

		if(printable == 1){
			printChessBoard(chessBoard);
```

```
            }

            if(maxCountOfKnightTour == numberOfKnightTours){
                  exit(0);
            }

            chessBoard[posInRow][posInColumn] = 0;
            knightTourCounter--;
            return;
      }

      int i;
      for(i=0 ; i<8 ; i++){
            if((posInRow + possibleknightMoveY[i] < chessBoardHeight) &&
(posInRow + possibleknightMoveY[i] >= 0) &&
            (posInColumn + possibleKnightMoveX[i] < chessBoardWidth) &&
(posInColumn + possibleKnightMoveX[i] >= 0) &&
            (chessBoard[posInRow + possibleknightMoveY[i]][posInColumn +
possibleKnightMoveX[i]] == 0)){
                  countKnightMovesFromOneCell(posInRow +
possibleknightMoveY[i], posInColumn + possibleKnightMoveX[i],
chessBoard);
            }
      }

      chessBoard[posInRow][posInColumn] = 0;
      knightTourCounter--;
      return;
}

void printChessBoard(int **chessBoard){

      int i,j;
      for(i=0 ; i<chessBoardHeight ; i++){
            for(j=0 ; j<chessBoardWidth ; j++){
                  printf("%d\t", chessBoard[i][j]);
            }
            printf("\n");
      }
      printf("\n\n");
}
```

**Лістинг А.3 – Вихідний програми на основі методу Варнсдорфа**

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <limits.h>

//functions
int** initTwoDimentionalArray(int columns, int rows);
void varnsdortFromAllCells(int height, int width, int **chessBoard);
int tourIsPossible(int width, int height, int currentPosY, int
currentPosX);
int varnsdorf(int height, int width, int **chessBoard, int
currentPosY, int currentPosX);
```

```
int countNumberOfAvailableMoves(int height, int width, int
**chessBoard, int currentPosY, int currentPosX);
void printChess(int height, int width, int **chessBoard);
void clearChess(int height, int width, int **chessBoard);

//variables
const int possibleKnightMoveX[] = { -1, -1, 1, 1, -2, -2, 2, 2 };
const int possibleKnightMoveY[] = { 2, -2, 2, -2, 1, -1, 1, -1 };

int main(int argc, char *argv[]) {

	int height, width;

	printf("Enter the height of chessboard: ");
	scanf("%d", &height);

	printf("Enter the width of chessboard: ");
	scanf("%d", &width);

	int **chessBoard = initTwoDimentionalArray(height, width);

	varnsdortFromAllCells(height, width, chessBoard);

	return 0;
}

int** initTwoDimentionalArray(int rows, int columns){

	int **array = (int**)calloc(rows, sizeof(int*));

	int i;
	for(i=0 ; i<rows ; i++){
		array[i] = (int*)calloc(columns, sizeof(int));
	}

	return array;
}

void varnsdortFromAllCells(int height, int width, int **chessBoard){

	int i,j;
	int numberOfPassedCells;
	for(i=0 ; i<height ; i++){
		for(j=0 ; j<width ; j++){
			if(tourIsPossible(width, height, i, j)){
				numberOfPassedCells = varnsdorf(height, width,
chessBoard, i, j);
				printf("\n\nStarting from [%d][%d]\n", i, j);
				printf("Number of passed cells: %d\n",
numberOfPassedCells);
				printChess(height, width, chessBoard);
				clearChess(height, width, chessBoard);
			}
		}
	}
}
```

```
int tourIsPossible(int width, int height, int currentPosY, int
currentPosX){
      if(((width * height)%2 != 0)
            && ((currentPosY + currentPosX)%2 != 0)){
            return 0;
      }
      return 1;
}

int varnsdorf(int height, int width, int **chessBoard, int
currentPosY, int currentPosX){

      int i,j;

      for(i=0 ; i<height*width ; i++){
            chessBoard[currentPosY][currentPosX] = i+1;

            int whereToGo;
            int endMove = -1;
            int minAvailableMoves = INT_MAX;
            for(j=0 ; j<8 ; j++){
                  if(possibleToMove(height, width, chessBoard,
currentPosY, currentPosX, j)){
                        endMove = j;
                        int numberOfAvailableMoves =
countNumberOfAvailableMoves(height, width, chessBoard,
currentPosY+possibleKnightMoveY[j],
currentPosX+possibleKnightMoveX[j]);
                        if(numberOfAvailableMoves>0 &&
numberOfAvailableMoves<minAvailableMoves){
                              minAvailableMoves = numberOfAvailableMoves;
                              whereToGo = j;
                        }
                  }
            }

            if(endMove != -1 && minAvailableMoves == INT_MAX){

      chessBoard[currentPosY+possibleKnightMoveY[endMove]][currentPosX+
possibleKnightMoveX[endMove]] = i+2;
                  return i+2;
            }else if(minAvailableMoves == INT_MAX){
                  return i;
            }

            currentPosY += possibleKnightMoveY[whereToGo];
            currentPosX += possibleKnightMoveX[whereToGo];
      }
}

int possibleToMove(int height, int width, int **chessBoard, int
currentPosY, int currentPosX, int i){

      if((currentPosY + possibleKnightMoveY[i] < height) &&
(currentPosY + possibleKnightMoveY[i] >= 0) &&
      (currentPosX + possibleKnightMoveX[i] < width) && (currentPosX +
possibleKnightMoveX[i] >= 0) &&
```

```
      (chessBoard[currentPosY + possibleKnightMoveY[i]][currentPosX +
possibleKnightMoveX[i]] == 0)){
            return 1;
      }

      return 0;
}

int countNumberOfAvailableMoves(int height, int width, int
**chessBoard, int currentPosY, int currentPosX){

      int i, counter=0;
      for(i=0 ; i<8 ; i++){
            if(possibleToMove(height, width, chessBoard, currentPosY,
currentPosX, i)){
                  counter++;
            }
      }

      return counter;
}

void printChess(int height, int width, int **chessBoard){

      int i,j;
      for(i=0 ; i<height ; i++){
            for(j=0 ; j<width ; j++){
                  printf("%d\t", chessBoard[i][j]);
            }
            printf("\n");
      }
}

void clearChess(int height, int width, int **chessBoard){

      int i,j;
      for(i=0 ; i<height ; i++){
            for(j=0 ; j<width ; j++){
                  chessBoard[i][j] = 0;
            }
      }
}
```