Звіт про Backtracking

9 травня 2025 р.

Вступ: Бектрекінг та його роль в дискретній математиці

Бектрекінг (від англ. *backtracking*) — це метод розв'язання комбінаторних задач, який базується на ідеї систематичного перебору всіх можливих рішень із можливістю відкату (повернення до попереднього стану), якщо поточний шлях не веде до цілі. Цей підхід дозволяє ефективно знаходити розв'язки у випадках, коли повний перебір був би надто ресурсомістким.

З формальної точки зору, бектрекінг є реалізацією пошуку в глибину (DFS) у просторі рішень, який може бути представлений у вигляді дерева або графа. Його основною перевагою є можливість «обрізати» гілки, які завідомо не ведуть до допустимого результату. Таким чином, це не просто сліпий пошук, а розумне обмеження простору можливостей за рахунок логічних перевірок.

Бектрекінг широко використовується в задачах пошуку з обмеженнями, таких як розміщення об'єктів (наприклад, ферзів на шахівниці), проходження лабіринтів, розв'язання судоку, генерація перестановок або підмножин, ігри з пошуком оптимального ходу. Його універсальність і логічна прозорість робить його одним із базових інструментів для вивчення прикладної дискретної математики та алгоритмів штучного інтелекту. Ми використали цей алгоритм в дії для виконання нижче викладених задач:

Sudoku

Цей модуль реалізує алгоритм розв'язання судоку довільного розміру з використанням методу бектрекінг. Додатково підтримується варіант з жадібним вибором клітинок, що оптимальніше. Модуль розбивається на файли sudoku_cls.py, sudoku_vis.py та __main__.py, перші два з який реалізують головні класи Sudoku та SudokuVis а в __main__.py використовується argparse для зручного запущення модулю.

Основні методи класу Sudoku:

- `solve(greedy=False, visualization=None, random_fill=False)` — основна рекурсивна функція для розв'язання Судоку. Використовує жадібну версію, якщо `greedy=True`, або заповнює числа у випадковому порядку, якщо `random_fill=True`. Якщо вказано `visualization` як клас `SudokuVis` то оновлює дошку на візуалізації
- `get_valid_numbers(row, col)` — повертає список можливих чисел для заданої клітинки за правилами судоку.
- `generate_cell_order(greedy=False)` — генерує впорядкований список клітинок для проходження. При жадібному режимі сортує їх за кількістю можливих значень.
- `fill(fill_chance)` — генерує заповнену дошку методом solve(random_fill=True) і випадково видаляє частину значень з вказаною ймовірністю `fill_chance`, що гарантує, що дошку можливо розв'язати.
- `is_board_valid()` — перевіряє чи дошка є валдіною.

Основні частини класу SudokuVis:

- Сам клас `SudokuVis(master, board)` — основний клас візуалізації. Ініціалізує графічний інтерфейс, приймаючи розмір або готову дошку. Автоматично визначає розмір блоків.
- `build_grid()` — створює графічну таблицю відповідно до вхідної дошки.

- `solve(greedy=False)` — запускає алгоритм розв'язання (опціонально жадібний), візуалізуючи кроки, якщо включено візуалізацію.

У гіршому випадку, складність алгоритму близько до $O(k^n)$, де k — кількість можливих чисел на клітинку, n — кількість порожніх клітинок. Жадібна оптимізація значно скорочує кількість рекурсивних викликів завдяки впорядкуванню клітинок.

Існує підтримка дошок довільного розміру, як і стандартної 9×9 так і 16×16 чи 18x18. Можливе створення пустої дошки, або з файлу чи наповненої за якимось відсотком. Є можливість візуалізації через графічний інтерфейс. Також можливе проведення тестувань з різними розмірами дошок, рівнями заповненості та кількостями ітерацій.

Жадібний алгоритм більшості часу виходить швидчий за звичайний backtracking, але це тяжко порівняти адже це сильно залежить від окремих згенерованих дошок (деякі можуть розв'язуватися до 15 хв).

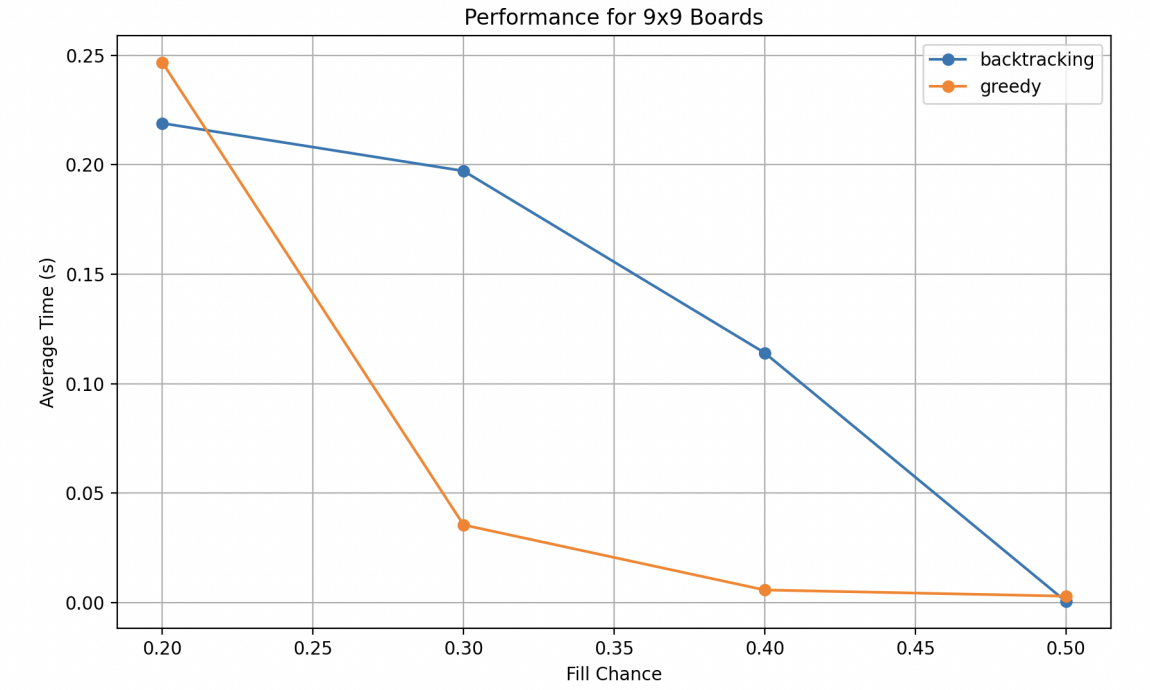


Рис. 1: Порівняння алгоритмів для 9x9 судоку

Maze Finder

Цей код реалізує алгоритм рекурсивного пошуку всіх можливих шляхів від точки "A"до точки "B"у лабіринті, представленому двовимірною матрицею. Стінки позначені числом 1, прохідні клітинки — 0, початкова та кінцева точки — символами "A"та "B"відповідно.

Основна логіка:

- `find_all_paths(maze, solution, start, final)` — рекурсивна функція, яка з використанням глибокого копіювання (deepcopy) відмічає відвідані клітинки та додає нову позицію до поточного шляху. Якщо знайдено точку "B шлях зберігається у список final. Алгоритм розглядає всі 4 напрямки руху (вгору, вниз, вліво, вправо) та гарантує, що не повертається до вже відвіданих клітинок.

  Процес:

  1. Генерується карта з стінами, входом і виходом у визначених місцях.
  2. Рух від старту відбувається в одну з кардинальних сторін (вгору, вниз, вліво, вправо).
  3. Якщо у вибраному напрямку немає стіни або краю лабіринту, рух триває рекурсивно.
  4. Якщо позиція досягає виходу, шлях записується як правильний.
  5. Серед усіх правильних шляхів обирається найкоротший, який підсвічується в кінці.

Складність алгоритму: у гіршому випадку — експоненційна, тобто $O(4^k)$, де k — кількість вільних клітинок (бо кожен крок може мати до 4 варіантів продовження шляху).

Також є можливість візуалізувати мапу лабіринту після запуску програми зі стінами, входами і виходами. **Порівняння:** Dfs себе показує завжди швидше в порівнянні з методом backtracking через те, що бектрекінг іде завжди в всі сторони по декілька разів

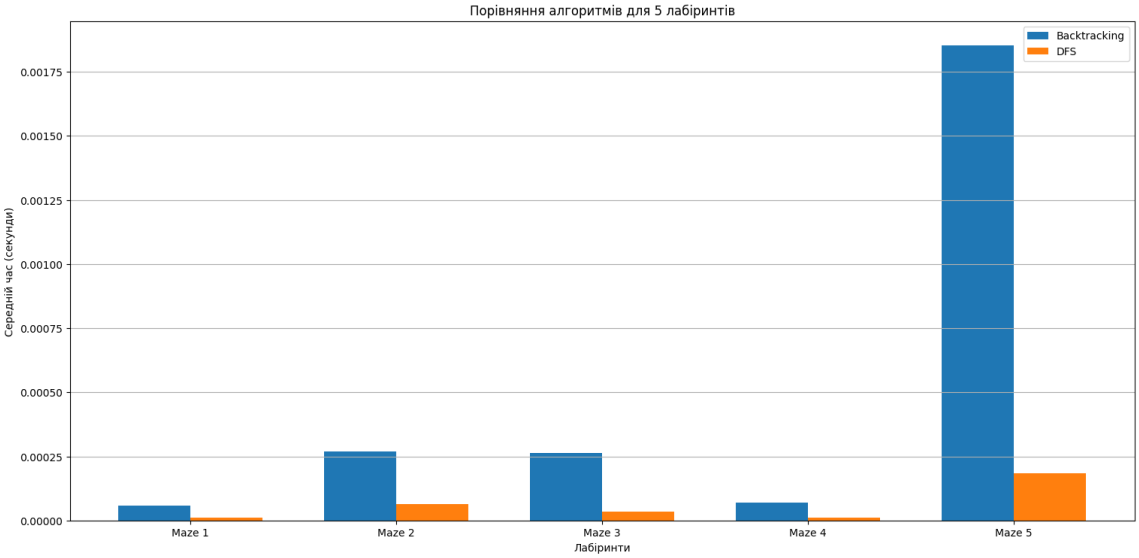


Рис. 2: Порівняння алгоритмів для 5 лабіринтів.

Crossword

Код шукає можливе заповнення кросворду словами, якщо не виходить – то повертається назад і шукає інший варіант. Кожен крок вставлення і прибирання слів відображається.

Клас `Crossword` – клас для реалізації задачі. Приймає аргументи графічний інтерфейс для візуалізації, список слів і сітку кросворду.

Метод `build_grid` створює сітку для візуалізації, а метод `update` оновлює візуалізацію, коли ми вписуємо або прибираємо слово.

Методи `place_horizontally` і `place_vertically` ставлять слово у визначену комірку.

Методи `remove_horizontally` і `remove_vertically` прибирають слово, якщо ми повертаємось назад.

Методи `can_place_horizontally` і `can_place_vertically` перевіряють чи є можливість поставити слово у горизонтальному або вертикальному положенні.

Метод `pre_start` підготовлює все для початку, перед бектрекінгом.

Метод `find_all_places` визначає всі комірки куди можна поставити слова.

Метод `crossword_backtrack` – це основна функція яка бектрекінгом намагається знайти розв'язок кросворду.

Функція `parse_args` дає можливість запустити код через термінал.

Приклад: [language=bash] python crossword.py –grid "1110111011,1010101110 ords "LIAR BUSTEE"

Сітка: `"1111111111,1010101010,1111111111"`

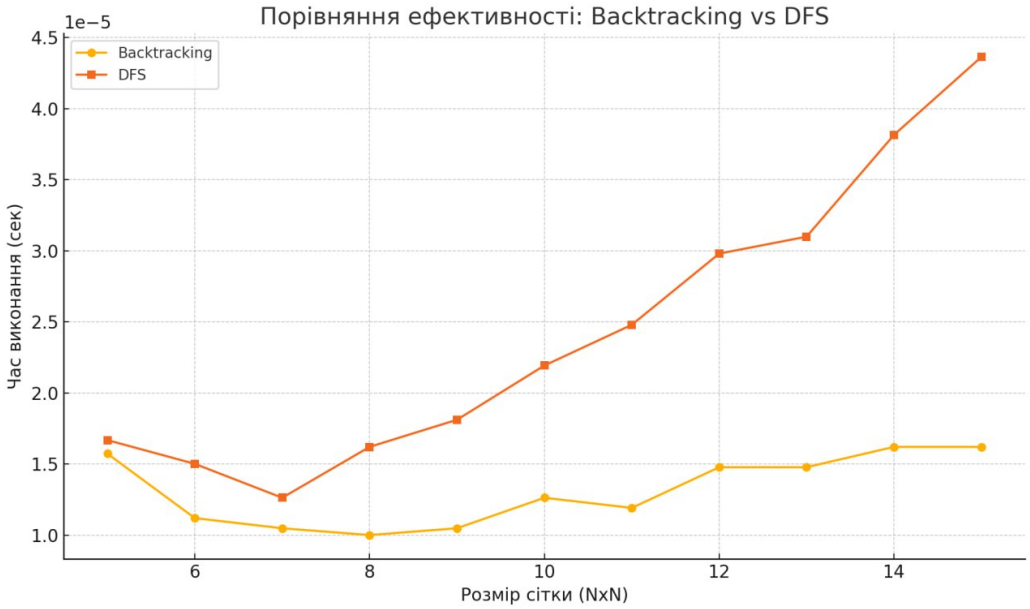
Слова: `["BUS "TEE "LESION"]`

Складність алгоритму: У гіршому випадку — експоненційна, оскільки ми пробуємо вставити всі слова в усі можливі позиції: $O(m^n), miii, nii.iiii, iii$.

Порівняння:

Dfs себе показує завжди повільніше в порівнянні з методом backtracking на великих кількостях даних.

beginfigure[H]



Queens

Цей код реалізує алгоритм знаходження всіх можливих способів розміщення n ферзів на дошці розміром rows × cols без конфліктів. Основна логіка побудована на методі *backtracking* — поетапному виборі позицій з можливістю відкату, якщо подальше розміщення стає неможливим.

Основні функції:

- `generate_board(queens, length, width)` — створює поточний стан дошки: на позиції ферзів ставить 'Q', а всі клітинки, які перебувають під атакою (по горизонталі, вертикалі та діагоналях), позначає як 1. Це допоміжна функція для візуального представлення.
- `find_all_queens(n, length, width, ...)` — рекурсивна функція, яка перебирає всі допустимі позиції на полі, слідкує за атакованими клітинками (через списки rows, cols, diag1, diag2), додає допустимі варіанти до фінального списку та повертається назад, щоб перевірити інші варіанти. Якщо використовується callback, виконується візуалізація проміжних кроків.

Процес:

1. Генерується дошка заданого розміру.
2. В позицію i, j ставиться ферзь.
3. Рекурсивно генерується нова дошка з ферзем, при цьому позначаються атаковані позиції як 1.
4. Якщо неможливо розмістити нового ферзя — повертається до попередньої позиції (бектрек).
5. Повертається список усіх комбінацій, у яких кількість ферзів перевищує n.

Складність алгоритму: $O(C(n,k))$ або наближено до $O(n!)$ для квадратної дошки. Алгоритм ефективно відсіює неможливі варіанти завдяки перевірці зайнятих рядів і діагоналей.

Порівняння:

Dfs себе показує завжди повільніше в порівнянні з методом backtracking на великих кількостях даних, але різниця не помітна і обидва алгоритми не оптимальні.

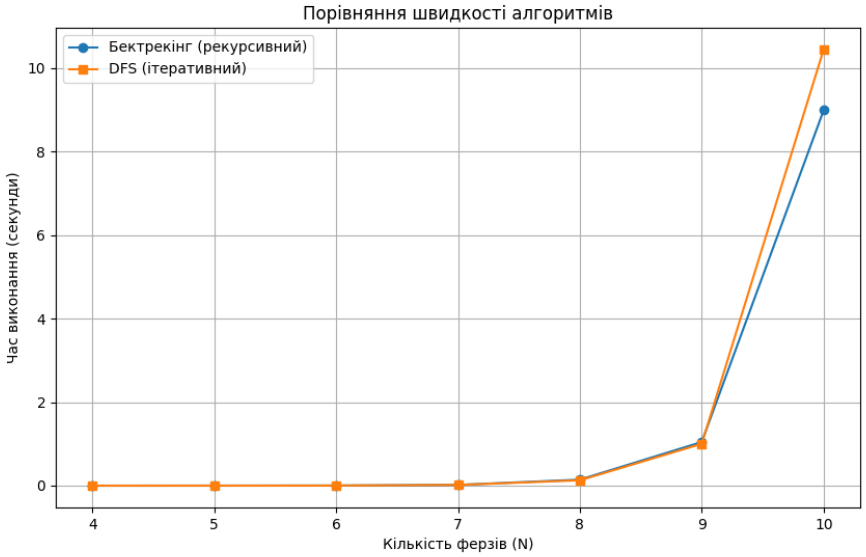


Рис. 3: Порівняння швидкості алгоритмів для задачі N ферзів.

K-coloring

Цей код реалізує алгоритм розфарбування графа в k кольорів — задачу, в якій необхідно присвоїти кожній вершині графа один з k кольорів так, щоб жодні дві суміжні вершини не мали однакового кольору. Основна логіка побудована на поетапному переборі кольорів з можливістю відкату, якщо поточне рішення веде до конфлікту. Весь функціонал реалізований в класі Graph. **Основні функції:**

- `generate(num_nodes, connectedness)` - генерує граф для розфарбування потім. Приймає як параметри num_nodes та connectedness. Перший визначає кількість вершин у графі, у другий - відсоток його повноти. Чим більше число - тим більш повним є граф, тобто тим більше в ньому ребер.
- `save()` - зберігає граф у форматі JSON до файлу graph_data.js.
- `render()` - зберігає граф як статичну картинку у за допомогою matplotlib.
- `k_coloring(k)` — функція, яка запускає алгоритм розфарбування. Містить в собі 3 допоміжні функції: can_color(node, color), update_history() step(index). Сама функція k_coloring() просто запускає допоміжну рекурсивну step(). В ній виконується весь алгоритм розфарбування.
- `can_color(node, color)` - Повертає, чи можна розфарбувати вершину в певний колір, перевіряючи всіх її сусідів на наявність серед них такого кольору.
- `update_history()` - додає поточне розфарбування до історії в файлі graph_data.js.
- `step(index)` - головна рекурсивна функція алгоритму, яка намагається призначити кольори вершинам графа одну за одною. Якщо всі вершини оброблені — повертає True (успішне розфарбування). Інакше перебирає можливі кольори для поточної вершини. Якщо жоден варіант не підходить, повертається назад (це і є backtracking).

Складність алгоритму: У загальному випадку — $\mathcal{O}(k^n)$, де n — кількість вершин. Проте завдяки перевірці на конфлікти між вершинами, алгоритм значно скорочує кількість непотрібних обчислень.

Розподіл роботи

Депутат Антон - queens, labyrinth і звіт

Гловін Максим - crosword і презентація
Рудько Адам - sudoku і реквайрменти
Омелянчук Кирило - k-coloring