Звіт: Пошук рішень з використанням алгоритму Backtracking

Команда №3: Бугір Єлизавета, Боднар Аліна, Карпіна Олеся, Дзюба Оксана

У цьому проєкті ми досліджуємо потужність алгоритму Backtracking для розв'язання класичних задач на пошук та комбінаторику. Ми реалізували декілька задач і проаналізували їх складність, ефективність порівняно з іншими методами розв'язання.

Задачі, які ми реалізували:

- Розв'язання судоку - Боднар Аліна
- Вихід з лабіринту - Карпіна Олеся
- Кросворд - Карпіна Олеся
- Задача розфарбування графа - Дзюба Оксана
- N-Queens - Бугір Єлизавета
- Візуалізація вище перелічених задач - Боднар Аліна

Реалізація на основі backtracking, порівняння ефективності з іншими методами, аналіз складності алгоритмів

Розв'язання судоку

Алгоритм Backtracking - це метод, який перебирає всі можливі варіанти і повертається назад, якщо вибраний шлях не веде до розв'язку.

Кроки:

1. У нашій реалізації він спочатку шукає першу порожню клітинку за допомогою `find_empty_cell`. Програма шукає першу клітинку, яка ще не заповнена (0). Якщо таких немає - судоку розв'язане, і функція повертає True.
2. Спроба вставити число від 1 до 9 для знайденої порожньої клітинки: `for num in range(1, size + 1):` Але ми вставляємо число лише якщо воно не порушує правила судоку.
3. Перевірка правильності вставки за допомогою `is_valid(board, row, col, num)`. Ця функція перевіряє:
   - чи вже є таке число у цьому рядку
   - чи є в цьому стовпці
   - чи є в маленькому 3×3 квадраті
4. Рекурсивний виклик для наступної клітинки. Після вставки алгоритм знову викликає сам себе для наступної порожньої клітинки: `if solve_backtracking(board): return True`
5. Якщо шлях неправильний — ми повертаємося назад (робимо "відкат"): `board[row][col] = 0`

Візуалізація

- Консоль: Використано модуль colorama, щоб підсвічувати клітинку, яку програма щойно заповнила:

```
from colorama import init, Fore, Back, Style
init(autoreset=True)

```

`highlight` підсвічує зеленим фоном клітинку, яка зараз заповнюється. Після кожного кроку робиться затримка: `time.sleep(delay)`

- Візуалізація через Pygame: Створюється вікно 540x540 пікселів. Кожна клітинка малюється у прямокутнику. Коли заповнюється нова клітинка — вона відображається зеленим кольором. Після кожного ходу робиться пауза `pygame.time.delay(100)` для наочності. Щоб запустити розв'язування або закрити вікно треба натиснути пробіл.

Запуск:

Запускається з терміналу командою: `python3 sudoku.py console`.
Щоб запустити візуальну версію: `python3 launcher.py`

Порівняння різних методів

Ми протестували різні імплементації судоку через використання бектрекінгу, dfs та жадібного методу.

Детальніше про методи:

- DFS (глибина у пошук) - перебирає варіанти послідовно
- Backtracking (з поверненням) - розумніше шукає рішення, повертається назад при помилках
- Greedy (жадібний) - ставить число в першу доступну клітинку, без перевірки всіх варіантів

Ми протестували алгоритми на двох розмірах дошок, стандартному 9x9 і більшому розмірі 16x16. Також ми використали різну кількість підсказок (clues), щоб порівняти, наскільки швидко кожний алгоритм справляється з дошкою в залежності від заповненості початкової дошки. Для дошки 9x9 було дано 20, 30 і 40 підсказок. Для 16x16 - 110, 120 і 130.
Ми вивели 3 графи, щоб порівняти ефективність цих методів:

- перший показує як змінюється час виконання усіх трьох методів залежно від розміру дошки;
- другий показує їхню роботу залежно від кількості підсказок на дошці 9x9
- третій показує їхню роботу залежно від кількості підсказок на дошці 16x16

Аналіз результатів

- **Greedy алгоритм** — найшвидший у всіх випадках:
  - Обчислювальна складність: $O(N^2)$
  - Практично завжди працює за 0.0001 - 0.0004 секунд, незалежно від розміру або кількості підказок
  - Як працює: Алгоритм заповнює порожні клітинки найпростішим способом: він бере перше можливе число, яке не порушує правила судоку в конкретному місці, і переходить до наступної клітинки. Він не перевіряє, що буде далі, і не повертається назад у разі помилки.
  - Чому він швидкий: Він не перевіряє варіанти наперед, а лише перший, що підходить. Також немає рекурсії або циклічного перебору варіантів.

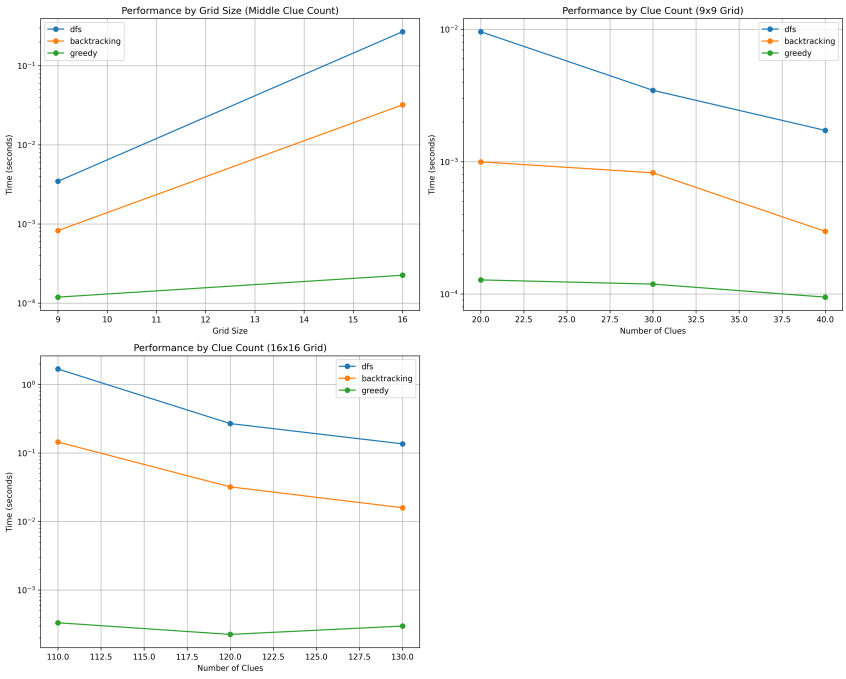

Рис. 1: Порівняння методів розв'язання судоку

  - Проте він не гарантує правильне чи унікальне рішення. Може виявитися, що подальші клітинки вже неможливо заповнити без конфліктів, але алгоритм цього не помічає.

- **Backtracking** — стабільний і досить швидкий:
  - Обчислювальна складність: $O(9^M) : i, ii.i, i, iiiii.$

– Чому швидший за DFS: Він використовує логіку для відсікання поганих рішень рано, тому не перебирає всі варіанти. Часто розв'язує задачу дуже швидко, особливо коли підказок достатньо (тобто задача не дуже складна).

– Чому повільніше за Greedy: Використовує рекурсію та перевірки кількох варіантів, для точного та гарантовано правильного розв'язання.

- **DFS** — найповільніший і менш передбачуваний:
  - Обчислювальна складність: $O(9^M) : iii, ii, i.i"", iii.$

– Чому він повільний: Повністю перебирає всі варіанти, навіть ті, які можна було би відкинути раніше. Погано працює з великими сітками або задачами з малою кількістю підказок (тобто високою складністю).

– На приклад, у випадку з сіткою 16x16 та 120 підказками, час DFS міг зростати до 2 секунд, тоді як Backtracking розв'язував те саме завдання за менше ніж 0.1 секунди.

Можна зробити висновок, що найкраще підходить backtracking: він майже завжди швидкий, і гарантує правильне рішення, на відміну від швидшого greedy методу.

Вихід з лабіринту

Реалізація рішення для задачі пошуку шляху в лабіринті з використанням бектрекінгу з евристичною оптимізацією.

Мета — знайти найкоротший шлях від стартової точки A до кінцевої B у складному середовищі.

Основні етапи:

Ми використовуємо рекурсивний пошук з поверненням (бектрекінг). У кожній точці лабіринту намагаємось перейти в напрямку, що веде до B, і позначаємо шлях. Якщо до B дійти не вдається — повертаємось назад, і пробуємо інший варіант.

1. Старт з початкової точки A.
2. Знаходимо координати A і B у лабіринті
3. Пошук сусідів: Отримуємо допустимі сусідні клітинки, куди можна рухатися (не виходячи за межі, не заходячи в стіни):
4. Сортування сусідів (евристика): Перевага надається тим напрямкам, які ближчі до точки B (Манхеттенська відстань):

```
neighbors.sort(key=lambda pos: abs(pos[0] - end[0]) + abs(pos[1] - end
    [1]))

```

5. Основна функція бектрекінгу:

```
def solve_maze(maze, row, col, path, best_path_length):
    if (row, col) == end:
        return True  #          '

    maze[row][col] = "*"  #
    for r, c in sorted_neighbors:
        if solve_maze(maze, r, c, path + [(r, c)], best_path_length):
            return True

    maze[row][col] = "."  #
    return False

```

Візуалізація:

- Консоль:
  - Вивід лабіринту з кольорами:
    - A — зелений
    - B — червоний
    - '*' — синій (поточний шлях)
    - . — сірий (відкат)
  - Затримка між кроками реалізована через time.sleep()
  - Очистка екрана через clear_screen()
- Pygame (графічна візуалізація):
  - Вікно 600x600 пікселів
  - Кожна клітинка відображається як прямокутник
  - Колір залежить від вмісту клітинки:
    - "A"— зелений
    - "B"— червоний
    - "*"— синій
    - "."— сірий
    - "#"— чорний (стіни)

Як запустити та вхід/вихід програми:

Режим задаємо через командний рядок:

- `python3 maze.py console` - консоль візуалізація
- `python3 maze.py visual` - pygame візуалізація

Лабіринт (maze) — жорстко заданий у коді. У режимі console – очікує натискання клавіші Enter для початку роботи.

Вихід:

- У режимі console:
  - Виводиться лабіринт у кольорах.
  - Відображається процес руху: символи '*' — крок уперед, '.' — шлях, який не привів до цілі.
  - Наприкінці — повідомлення:
    - Path found! Length: 20
    - або No path found!
- У режимі visual (pygame):
  - Відображається лабіринт у графічному вікні.
  - Зеленим — старт A, червоним — ціль B, синім — шлях *, сірим — помилкові кроки ..
  - Консоль повідомляє результат:
    - Path found! Length: 20
    - або No path found!

Порівняння з DFS, GREEDY:

Алгоритми оцінювались за двома критеріями: Кількість кроків та час виконання.

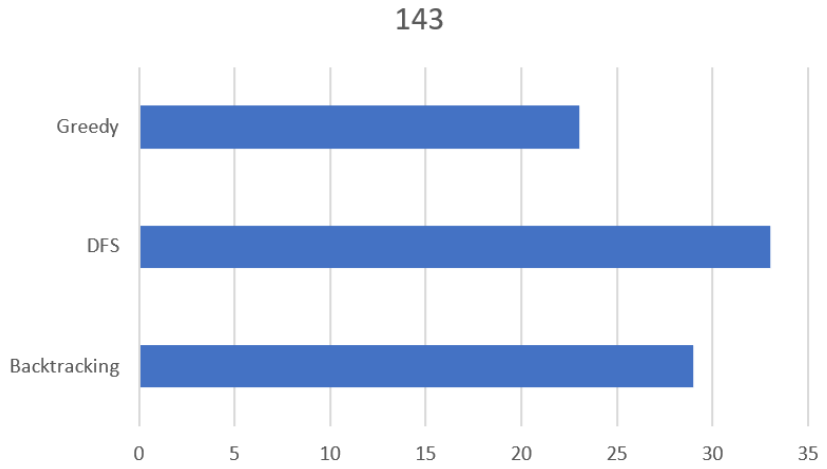

Рис. 2: Порівняння методів для лабіринту

Алгоритм Greedy є найшвидшим серед усіх трьох. У кращих випадках його складність становить O(n). Завдяки своїй простоті, Greedy швидко знаходить шлях, проте це не завжди оптимальне рішення, і іноді алгоритм взагалі не знаходить вихід у заплутаних структурах.

Складність DFS складає O(V + E), де V — кількість вершин у графі, а E — кількість ребер (у випадку гріду це приблизно $O(n^2)$). DFS гарантує знаходження рішення, якщо шлях існує, проте знайдений маршрут не завжди буде найкоротшим. У тестах DFS показав більшу кількість кроків і більший час виконання порівняно з Greedy, оскільки переглядає більше варіантів.

На практиці Backtracking працює значно швидше завдяки ранньому відсіканню невдалих гілок. У тестах Backtracking показав менше кроків, ніж DFS, і час виконання на тому ж рівні, що свідчить про його більш обережний і систематичний підхід до пошуку шляху. Backtracking демонструє хороший баланс між повнотою пошуку та ефективністю, що робить його надійним вибором для завдань, де важливо гарантовано знайти рішення навіть у складних лабіринтах.

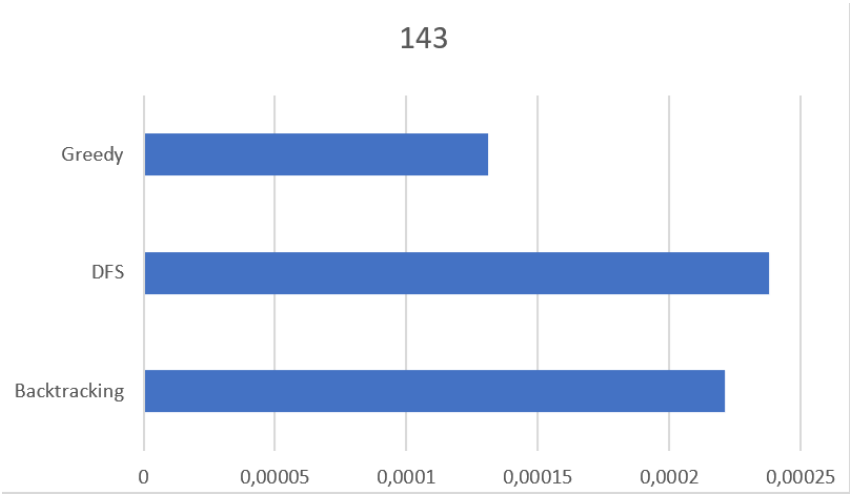

Рис. 3: Порівняння методів для лабіринту

Кросворд

Мета: автоматично заповнити кросворд заданими словами.

Основні етапи:

1. Завантаження слів: Слова беруться з файлу words_2.txt випадковим чином.
2. Початок рекурсії (`python метод solve класу BacktrackingSolver`):
   - Пробуємо розмістити перше слово у всіх можливих комірках по горизонталі та вертикалі.
   - Якщо слово підходить (`перевірка в is_valid_placement`):
     - Розміщуємо його на полі (`place_word`).
     - Малюємо поточний стан (візуально або в консолі).
     - Переходимо до наступного слова (`solve(index + 1)`).
3. Перевірка перетинів слів (`check_intersections`): Програма перевіряє, чи утворені нові слова при перетині входять у список допустимих.
4. Бектрекінг:
   - Якщо розмістити всі слова не вдається — останнє слово видаляється (`remove_word`) і пробується інша позиція.
   - Цей процес повторюється до тих пір, поки не знайдеться рішення або всі варіанти не буде вичерпано.

Візуалізація:

- Консоль:
  - '.' — порожня клітинка.
  - '#' — заблокована клітинка.
  - '[A]' — підсвічені перетини.
  - A — звичайні літери.
- Pygame (графічна візуалізація):
  - Білий — порожня клітинка для заповнення ('-').
  - Сірий — заповнена літера.
  - Жовтий — підсвічена комірка (напр. при новій спробі розміщення слова).
  - Чорний — рамки клітинок.
  - Перетини слів виділяються спеціально (у консолі — в квадратних дужках [A]).

Як запустити та вхід/вихід програми:

- Вхід:
  - Грід кросворду — список списків символів:
    - ’-’ — порожня клітинка для слова.
    - ’#’ — заблокована клітинка (сюди слова не ставляться).
  - Грід задається прямо у коді (змінна grid).
  - Слова — беруться з файлу words_2.txt (випадково вибирається, наприклад, 5 штук).
- Вихід:
  - Заповнений кросворд зі словами.
  - У візуальному режимі — показ процесу заповнення.
  - У консольному режимі — текстовий вивід поля після кожного кроку.
  - Повідомлення про успіх або неможливість знайти рішення.

Режим задаємо через командний рядок:

- `python3 crossword.py console` - консоль візуалізація
- `python3 crossword.py visual` - pygame візуалізація

Порівнюємо із greedy algorithm та brute force:

Шкала часу на графіку - логарифмічна. Порівняння роботи алгоритмів було виконано на 3 різних "дошках": 4x7, 13x7, 29x7.

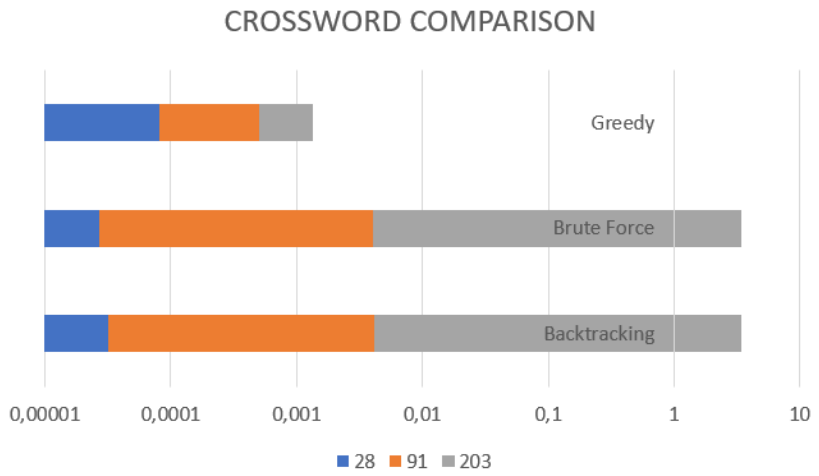

Рис. 4: Порівняння методів для кросворду

Жадібний алгоритм (Greedy) є найшвидшим для всіх розмірів сіток, проте важливо зазначити, що він не завжди знаходить рішення. Наприклад, на тестових сітках алгоритм впорався лише з найменшою з них, а на інших — завершувався помилкою, не змігши заповнити сітку повністю. Це відбувається тому, що жадібний підхід завжди обирає локально оптимальний варіант, не враховуючи глобальної картини.

Повний перебір (Brute Force) та бектрекінг (Backtracking) демонструють приблизно однакову швидкість, оскільки обидва мають експоненційну складність $O(p^w), p^-iii, w^-iiii.i, iiii, ii.$

Задача розфарбування графа

Задача розфарбування графа полягає в тому, щоб присвоїти кольори вершинам графа так, щоб жодні дві суміжні вершини не мали однакового кольору. Це класична задача теорії графів.

У цьому проєкті реалізовано алгоритм бектрекінгу для пошуку допустимого розфарбування графа. Реалізація підтримує два режими: консольний і візуальний (з використанням matplotlib).

Кроки алгоритму

1. Ініціалізація:
   - Створюється матриця суміжності графа.
   - Задається максимальна кількість дозволених кольорів (m_colors).
2. Рекурсивне розфарбування вершин:
   - Починаючи з вершини 0, алгоритм намагається призначити їй колір від 1 до m_colors.
   - Для кожного кольору перевіряється, чи можна його безпечно призначити (без конфлікту з сусідами).
3. Перевірка коректності кольору:
   - Якщо хоча б одна суміжна вершина вже має цей колір — поточний колір не підходить.
   - Якщо конфлікту немає — колір зберігається, і запускається розфарбування наступної вершини.
4. Повернення назад (backtracking):
   - Якщо для поточної вершини неможливо знайти жодного допустимого кольору, алгоритм повертається до попередньої вершини та пробує інший варіант кольору.
5. Завершення:
   - Якщо всі вершини успішно розфарбовано — алгоритм завершується з успіхом.
   - Якщо ні — виводиться повідомлення про відсутність можливого розфарбування з заданою кількістю кольорів.

Візуалізація

- Консольна версія:
  - Виводить поточний стан розфарбування після кожного призначення або відкату.
  - Використовуються текстові назви кольорів (Gray, Teal, Green тощо) для кращої читабельності.
  - Дозволяє стежити за логікою виконання backtracking.
- Графічна версія (matplotlib):
  - Будує граф з допомогою бібліотеки networkx.
  - Вершини розфарбовуються відповідно до призначених кольорів.
  - На кожному кроці показується новий стан графа (включаючи кроки відкату).
  - Використовується plt.pause(0.5) для імітації "анімації"процесу.

Запуск програми

Програма підтримує запуск у двох режимах через аргумент командного рядка:

- `python script.py console` - запускає текстовий режим.
- `python script.py visual` - запускає графічну візуалізацію процесу.

Аналіз результатів

Для оцінки ефективності реалізованого алгоритму розфарбування графа методом бектрекінгу, проведено порівняння з двома іншими підходами DFS та Greedy. На графіках зображено залежність часу виконання алгоритмів від кількості вершин у графі (від 5 до 10).

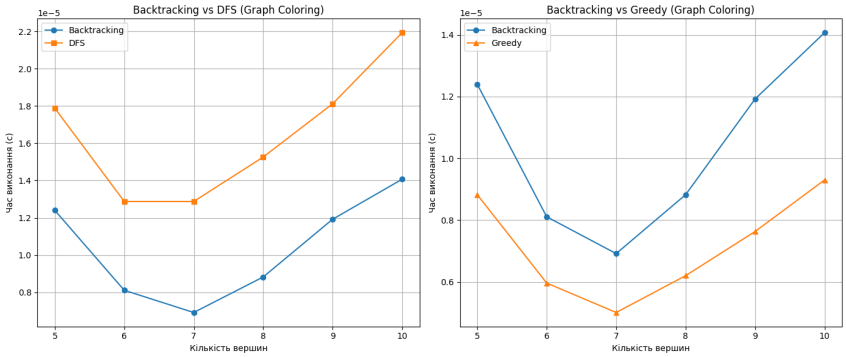

Рис. 5: Порівняння методів для задачі розфарбування графа

Порівняння Backtracking і DFS

- Алгоритм DFS показує стабільно вищий час виконання у всіх випадках, особливо при збільшенні кількості вершин.
- У випадку 10 вершин, DFS потребує майже вдвічі більше часу, ніж backtracking.
- Це пояснюється тим, що DFS не враховує обмеження кольорів під час обходу, що призводить до додаткових перевірок і гіршої оптимізації.

Порівняння Backtracking і Greedy

- Жадібний алгоритм виявився найшвидшим серед трьох.
- Його час виконання менший за backtracking майже вдвічі при максимальній кількості вершин.
- Greedy приймає локальні оптимальні рішення без повного перебору варіантів, що знижує обчислювальні витрати, хоча іноді може використовувати більше кольорів, ніж потрібно.

Висновок

Алгоритм бектрекінгу забезпечує правильне і повне розфарбування графа, однак поступається в продуктивності жадібному підходу. У порівнянні з DFS, backtracking виявляється не лише точнішим, але й ефективнішим. Найшвидшим для невеликих графів виявився жадібний алгоритм, однак його варто використовувати лише там, де кількість кольорів не є критично важливою. Таким чином, вибір алгоритму залежить від балансу між точністю та швидкістю його дії, в залежності від того, яка задача перед нами стоїть.

N-Queens

N-Queens — це класична комбінаторна задача, де потрібно розмістити N ферзів на шаховій дошці розміром N×N так, щоб жоден ферзь не атакував іншого. Ферзь може атакувати по горизонталі, вертикалі та діагоналях.

Кроки:

1. **Пошук порожньої клітинки**: Алгоритм спочатку шукає порожню позицію на дошці (рядок, де ще немає ферзя).
2. **Спроба розмістити ферзя**: Для знайденого рядка перебираються всі можливі колонки, щоб спробувати розмістити ферзя.
3. **Перевірка коректності розміщення**: Коли позиція знайдена, перевіряється, чи не атакує ферзь інших ферзів (метод `_is_safe`).

4. **Рекурсивний виклик для наступного рядка**: Якщо позиція безпечна, рекурсивно розв'язується та сама задача для наступного рядка.

5. **Повернення назад (відкат)**: Якщо жодне положення в рядку не підходить або подальші кроки не приводять до розв'язку, алгоритм повертається до попереднього рядка і пробує інші варіанти: `board[row] = -1`.

Візуалізація

Консольна версія:

- Відображає рішення у вигляді сітки із символами `"Q"` для ферзів і `"."` для порожніх клітинок.
- Виводить статистику виконання алгоритму: кількість кроків, відкатів та час виконання.

Графічна версія (Tkinter):

- Створює вікно з шаховою дошкою розміром N×N.
- Дозволяє налаштувати розмір дошки та швидкість анімації.
- Візуалізує процес розміщення ферзів, підсвічуючи клітинки.
- Дозволяє переглядати різні розв'язки за допомогою кнопок навігації.
- Показує повну статистику виконання.

Порівняння різних методів

Ми протестували різні алгоритми розв'язання задачі N-Queens:

Детальніше про методи:

- **Backtracking** – розумно перебирає варіанти з поверненням при помилках:
  - Обчислювальна складність: $O(N!)$
  - Працює шляхом послідовного розміщення ферзів і відкату при конфліктах
  - Повертається назад і пробує інші варіанти, коли знаходить неправильний шлях
- **BFS (пошук у ширину)** – будує дерево всіх можливих розміщень:
  - Обчислювальна складність: $O(N!)$
  - Використовує чергу для зберігання часткових розміщень
  - Розширює всі можливі стани на кожному рівні, перш ніж перейти до наступного
- **Greedy (жадібний метод)** – обирає перші підходящі варіанти:
  - Обчислювальна складність: $O(N^2)$
  - Намагається розміщувати ферзів випадковим чином серед допустимих позицій
  - Не гарантує знаходження розв'язку, але працює дуже швидко

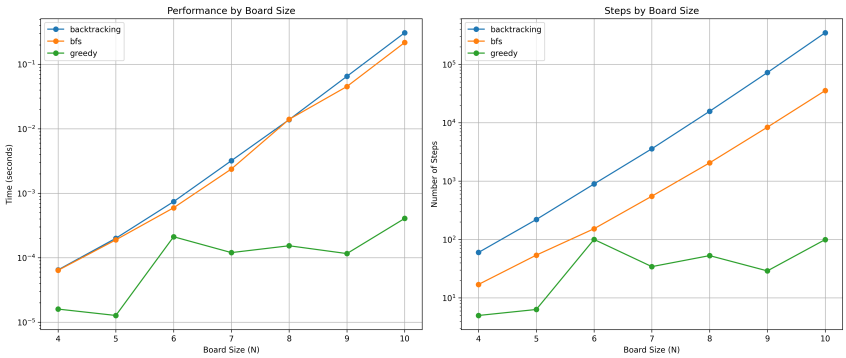

Рис. 6: Порівняння методів для задачі n queens

Аналіз результатів

Greedy алгоритм — найшвидший у всіх випадках:

- Час виконання: 10^{-5}–10^{-3} секунд (у 10–100 разів швидше за інші методи)
- Кількість кроків: від 5 до 300 (експоненційно менше порівняно з іншими)
- Відсутність гарантії знаходження розв'язку; хоча для N-Queens він часто знаходить хоча б один розв'язок
- Нестабільна продуктивність (видно на графіку "кроків де лінія має нерегулярний характер)

Backtracking — стабільний і досить ефективний:

- Час виконання: зростає експоненційно від 10^{-4} до 10^{-1} секунд для N від 4 до 10
- Кількість кроків: зростає від ~ 70 до $\sim 10^6$ при збільшенні N
- Гарантує знаходження всіх можливих розв'язків
- Передбачуване зростання складності (прямі лінії на логарифмічній шкалі)

BFS — масштабується гірше при зростанні N:

- Час виконання: близький до backtracking для малих N, але швидше стає непрактичним для $N > 10$
- Кількість кроків: від ~ 20 до $\sim 10^5$ для N від 4 до 10
- Використовує більше пам'яті через необхідність зберігати всі часткові стани у черзі
- Менш ефективний, ніж backtracking для цієї задачі через характер простору пошуку

Практичні висновки

Для малих розмірів дошки ($N \leq 8$):

- Всі алгоритми працюють достатньо швидко
- Greedy може бути найкращим вибором, якщо потрібен лише один розв'язок
- Backtracking найкраще підходить, якщо потрібні всі розв'язки

Для середніх розмірів ($8 < N \leq 15$):

- Greedy залишається дуже швидким, але може не знайти розв'язок з першої спроби
- Backtracking ще практичний, але час виконання значно зростає
- BFS стає непрактичним через обмеження пам'яті

Для великих розмірів ($N > 15$):

- Тільки Greedy може дати швидкий результат, але без гарантій
- Можливо потрібно застосовувати інші спеціалізовані методи

Оптимізації та можливі покращення

Для Backtracking:

- Оптимізація порядку вибору клітинок (наприклад, спочатку заповнювати клітинки з найменшою кількістю доступних варіантів)
- Використання бітових масок для перевірки допустимості розміщення

Для BFS:

- Використання евристики для пріоритезації найбільш перспективних часткових розв'язків
- Застосування пруніння станів для зменшення простору пошуку

Для Greedy:

- Впровадження локального пошуку або випадкових рестартів для підвищення ймовірності знаходження розв'язку
- Комбінування з іншими методами для отримання гарантованого розв'язку

Запуск програми

Консольна версія:

```
python3 nqueens.py console
```

У консольній версії користувачеві пропонується:

1. Ввести розмір шахової дошки (n)
2. Програма розв'язує задачу і виводить статистику (кількість рішень, час виконання, кроків, відкатів)
3. Користувач може переглянути знайдені розв'язки, вказавши кількість рішень для показу

Графічна версія (Tkinter):

```
python3 nqueens.py visual
```

У графічній версії користувач може:

- Вибрати розмір дошки з випадаючого списку (від 4 до 20)
- Налаштувати швидкість анімації ("Slow "Medium "Fast "Very Fast")
- Натиснути кнопку "Solve"для початку розв'язування
- Переглянути процес розміщення ферзів в реальному часі
- Використовувати кнопки навігації для перегляду різних розв'язків
- Бачити повну статистику виконання

Порівняльний аналіз алгоритмів:

```
python3 nqueens_comparison.py
```

Запускає бенчмарк, який:

- Порівнює продуктивність алгоритмів backtracking, BFS та greedy для різних розмірів дошки
- Генерує графіки часу виконання та кількості кроків

- Зберігає результати у файл $\texttt{nqueens}_b enchmark.png$

Висновок

Задача N-Queens є чудовим прикладом комбінаторної оптимізації, де можна порівняти різні підходи до пошуку розв'язків. Кожен метод має свої переваги та недоліки:

- Backtracking є найбільш універсальним і надійним, гарантуючи знаходження всіх розв'язків, але повільнішим для великих N.
- BFS схожий на backtracking за часом виконання, але використовує більше пам'яті та гірше масштабується.
- Greedy є найшвидшим, але стохастичним і не гарантує знаходження розв'язку.

Вибір алгоритму залежить від конкретних вимог: якщо потрібні всі розв'язки або гарантія знаходження хоча б одного — варто використовувати backtracking; якщо швидкість критична і достатньо одного розв'язку — greedy може бути кращим вибором.

Висновки: придатність алгоритму backtracking для різних класів задач

Алгоритм є придатним для різного типу задач та виправдовує свою ефективність, точність.