Звіт до лабораторної роботи

Ярослав Грунда
Фі-21, ФТІ КПІ

15 вересня 2024 р.

Зміст

1 Мета роботи

Опанувати способи представлення та їх ефективної реалізації, проаналізувати швидкість роботи операцій над множинами у заданій реалізації - за допомогою двійкових векторів.

2 Опис структури класу `BitVectorSet`

Клас `BitVectorSet` реалізує множину за допомогою бітових векторів. Основна ідея полягає в тому, щоб зберігати множину елементів у вигляді бітових масивів, де кожен біт представляє наявність або відсутність елемента в множині. Кожен елемент множини відповідає певній позиції біта у 64-бітному регістрі.

2.1 Основні характеристики:

- **Регістри та універсум:**
  - Множина складається з t 64-бітних регістрів, що дозволяє зберігати до $64t$ елементів.
  - Універсум для цієї множини є діапазоном від 0 до $64t - 1$.
- **Представлення даних:**
  - Множина зберігається у вигляді списку `vector`, який містить t елементів. Кожен елемент цього списку є 64-бітним цілим числом, де біти відображають наявність елементів у множині.
- **Індексація та позиція бітів:**
  - Для кожного елемента множини за допомогою методу `_get_index` обчислюється, в якому 64-бітному регістрі цей елемент знаходиться, і яка його точна позиція у регістрі.
  - `register_index = element » 6`
    Ця операція використовує побітовий зсув вправо на 6 позицій (`» 6`) для обчислення індексу 64-бітного регістра, в якому знаходиться даний елемент. Оскільки $64 = 2^6$, нам потрібно змістити бінарне представлення числа на 6 бітів, щоб "відкинути"молодші біти, які відповідають позиції всередині регістра. Це еквівалентно цілочисельному діленню на 64:
    $$\text{register\_index} = \left\lfloor \frac{\text{element}}{64} \right\rfloor$$
    Ця операція дає індекс 64-бітного регістра, в якому знаходиться елемент.
  - `bit_position = element & (64 - 1)`
    Ця операція використовує побітове І (`&`) з числом 63 (тобто `64 - 1`, яке в двійковій формі виглядає як 111111). Це дозволяє отримати тільки молодші 6 біт числа `element`, що відповідають позиції біта всередині 64-бітного регістра. Іншими словами, операція `element & 63` повертає залишок від ділення елемента на 64:
    $$\text{bit\_position} = \text{element}\%64$$

Таким чином, ця операція дає позицію біта всередині 64-бітного регістра.

2.2 Методи класу `BitVectorSet`

- `__init__(self, t)`:
  - Ініціалізує порожню множину, яка складається з t 64-бітних регістрів. Це означає, що універсум для цієї множини має $64t$ елементів.
  - Складність: $O(t)$, оскільки створюється список з t регістрів, кожен з яких ініціалізується до нульового значення.
- `_get_index(self, element)`:
  - Обчислює індекс регістра та позицію біта для заданого елемента.
  - `register_index = element » 6`
    Ця операція використовує побітовий зсув вправо на 6 позицій, що відповідає цілочисельному діленню на 64. Складність: $O(1)$.
  - `bit_position = element & (64 - 1)`
    Ця операція використовує побітове І з числом 63, щоб отримати позицію біта в регістрі. Складність: $O(1)$.
- `have(self, element)`:
  - Перевіряє наявність елемента в множині, повертаючи `True` або `False`.
  - Складність: $\Theta(1)$, оскільки перевірка бітового масиву займає константний час.
- `add(self, element)`:
  - Додає елемент до множини, встановлюючи відповідний біт у відповідному регістрі.
  - Складність: $\Theta(1)$, оскільки операція побітового OR є константною за часом.
- `delete(self, element)`:
  - Видаляє елемент з множини, скидаючи відповідний біт у регістрі.
  - Складність: $\Theta(1)$, оскільки операція побітового AND з інверсією є константною за часом.
- `union(self, other)`:
  - Повертає нову множину, яка є об'єднанням двох множин (логічна операція OR для кожного регістра).
  - Складність: $O(t)$, оскільки потрібно виконати операцію OR для кожного регістра.
- `intersection(self, other)`:
  - Повертає нову множину, яка є перетином двох множин (логічна операція AND для кожного регістра).

  - Складність: $O(t)$, оскільки потрібно виконати операцію AND для кожного регістра.

- `diff(self, other)`:
  - Повертає різницю між двома множинами (елементи, які є в першій множині, але відсутні в другій).
  - Складність: $O(t)$, оскільки потрібно виконати операцію AND з інверсією для кожного регістра.

- `sym_diff(self, other)`:
  - Повертає симетричну різницю між двома множинами (елементи, які є в одній з множин, але не в обох одночасно).
  - Складність: $O(t)$, оскільки потрібно виконати операцію XOR для кожного регістра.

- `issubset(self, other)`:
  - Перевіряє, чи є поточна множина підмножиною іншої.
  - Складність: $O(t)$, оскільки для кожного регістра перевіряється, чи всі біти з першої множини присутні у другій.

- `clear(self)`:
  - Очищає множину, встановлюючи всі біти у 0.
  - Складність: $O(t)$, оскільки потрібно оновити всі регістри.

- `__str__(self)`:
  - Повертає рядок, який представляє бітовий вектор множини для зручного перегляду. Регістри відображаються у зворотному порядку.
  - Складність: $O(t)$, оскільки потрібно конвертувати кожен регістр у рядок і об'єднати їх.

3 Тестування швидкості

3.1 Результати методу `union`

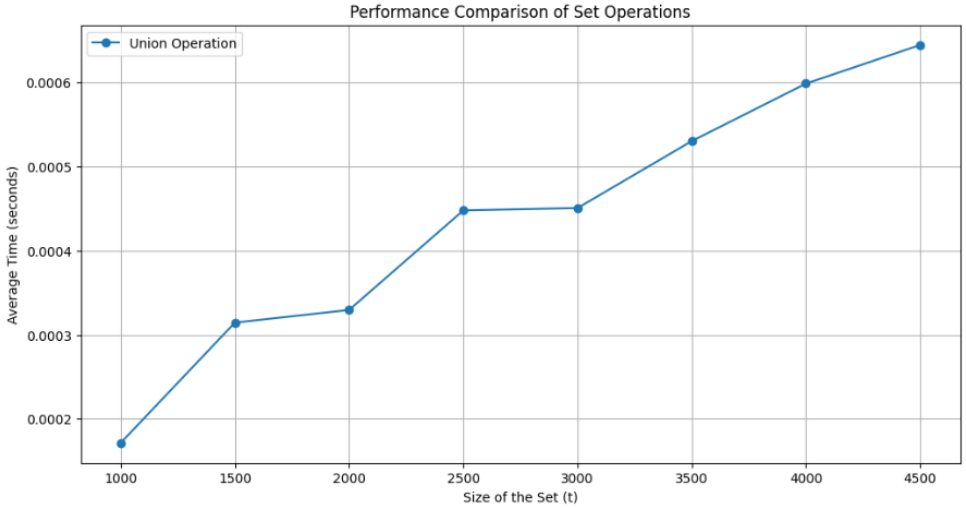

Рис. 1: Порівняння продуктивності операції об'єднання (`union`) для різних розмірів множин. По горизонтальній осі відкладено розмір множини (t), а по вертикальній — середній час виконання операції у секундах.

На графіку представлено порівняння продуктивності операції об'єднання (`union`) для різних розмірів множин. По горизонтальній осі відкладено розмір множини (t), а по вертикальній — середній час виконання операції у секундах.

Аналіз результатів: Як видно з графіку, час виконання операції об'єднання поступово зростає зі збільшенням розміру множини. Для невеликих множин (розмір близько 1000 елементів) операція займає приблизно 0.0001 секунди. Однак із збільшенням кількості елементів до 4500 середній час виконання операції підвищується до 0.0007 секунди.

Це свідчить про те, що операція `union` демонструє лінійну залежність від розміру множини, що можна пояснити її алгоритмічною складністю. Дані підтверджують, що зі зростанням розміру вхідних даних кількість обчислень і, відповідно, час виконання операції зростають майже пропорційно.

3.2 Результати методу `search`

Аналіз результатів: Оранжева лінія (пошук неіснуючого елемента) демонструє значні коливання в часі виконання. Наприклад, на множинах розміром близько 2500 елементів спостерігається пікове зростання часу до 4 мікросекунд, після чого час пошуку різко знижується до 0. Це свідчить про непостійний час виконання цієї операції, який може залежати від структури даних, розташування елементів у пам'яті або інших системних факторів.

Синя лінія (пошук існуючого елемента) показує більш стабільну поведінку. На розмірах 1000-3000 час виконання знаходиться в межах 1–2 мікросекунд і має незначні коливання. Після 3000 час падає близько до 0, що може свідчити про оптимізацію або стабільну продуктивність у цих умовах.

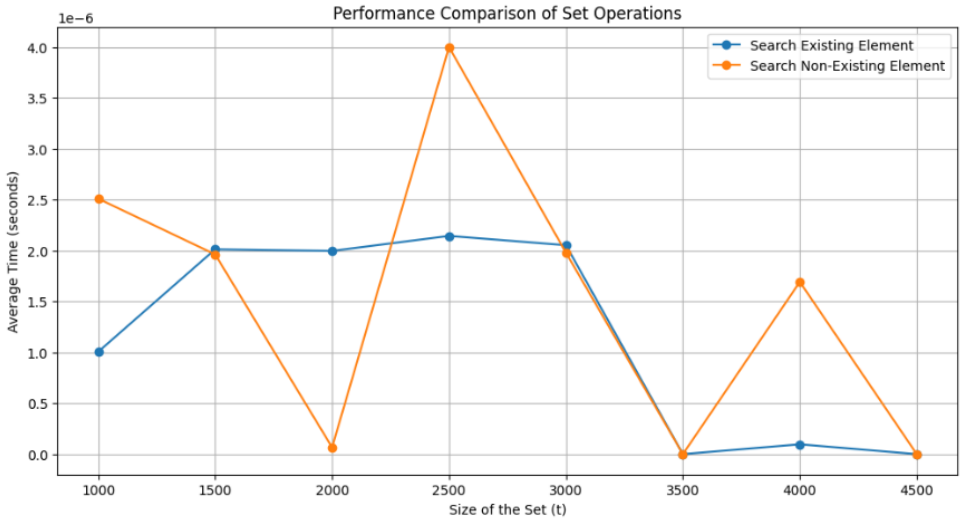

Рис. 2: Порівняння продуктивності операцій пошуку існуючих та неіснуючих елементів у множині різного розміру. По горизонтальній осі відкладено розмір множини (t), а по вертикальній — середній час виконання операції у секундах (в масштабі e^{-6}).

Проте незважаючи на всі коливання, ці операції залишаються однозначно швидкими, оскільки масштаб e^{-6} говорить про майже моментальне виконання незважаючи на розмір множини.

4 Висновки

Структура, реалізована за допомогою бітових векторів, є ефективною для операцій перевірки наявності, додавання та видалення елементів у множині. Це досягається завдяки константному часу виконання цих операцій.

Для більш детальної інформації, будь ласка, відвідайте наступний ресурс: Перейти до репозиторію.