Київський національний університет імені Тараса Шевченка
Факультет комп’ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Алгоритми та складність

Лабораторна робота №2-1

“Реалізація оптимального бінарного дерева пошуку

(динамічне програмування)”

Виконала студентка 2-го курсу

Групи ІПС-21

Сенечко Дана Володимирівна

Київ - 2025

Завдання

Реалізувати оптимальне бінарне дерево пошуку (динамічне програмування). Тип даних за варіантом Т7 - комплексні числа.

Теорія

Оптимальне бінарне дерево пошуку - це таке бінарне дерево, яке мінімізує середню кількість порівнянь, необхідних для пошуку певного ключа у дереві. Це дерево поміщає ключі з більшою ймовірністю пошуку ближче до кореня дерева, тоді як ключі з нижчою ймовірністю пошуку дальше від дерева.

У двійковому дереві пошук деяких елементів може відбуватися частіше, ніж інших, тобто існують ймовірності p(k) пошуку k-го елемента і для різних елементів ці ймовірності не однакові. Можна припустити, що пошук в дереві в середньому буде швидшим, якщо ті елементи, які шукають частіше, будуть перебувати ближче до кореня дерева.

Нехай дано 2n+1 ймовірностей p_1, p_2 , ..., p_n , q_0, q_1 , ..., q_n , де p_i - ймовірність того, що аргументом пошуку є ki-елемент; qi -ймовірність того, що аргумент пошуку лежить між вершинами k_i і k_{i+1} ; q_0 - ймовірність того, що аргумент пошуку менше, ніж значення елемента k_1 ; q_n - ймовірність того, що аргумент пошуку більше, ніж k_n . Тоді ціна дерева пошуку C буде визначатися таким чином:

$$C = \sum_{j=1}^{n} p_j(levelroot_j + 1) + \sum_{k=1}^{n} q_k(levellist_k),$$

де $levelroot_j$ - рівень вузла, а $levellist_k$ - рівень листа k. Дерево пошуку називається оптимальним, якщо його ціна мінімальна. Тобто оптимальне

бінарне дерево пошуку – це бінарне дерево пошуку, побудоване в розрахунку на забезпечення максимальної продуктивності при заданому розподілі ймовірностей пошуку необхідних даних.

Алгоритм

Алгоритм побудови оптимального бінарного дерева пошуку ґрунтується на принципах **динамічного програмування** і полягає в пошуку мінімальної вартості дерева для підмножини ключів із заданими ймовірностями звернень до них.

1. **Вхідні дані**: ключі та їх ваги.

   Ключі сортуються за зростанням їх значень для спрощення пошуку оптимальної структури дерева. (В даному випадку для комплексних чисел рахуємо їх модуль).

2. **Динамічне програмування**: Створюється двовимірний масив $cost[i][j]$, який зберігає мінімальну вартість побудови дерева для елементів з індексами від i до j. Це вартість побудови піддерева з елементів від i до j включно.
   Вартість для кожного піддерева визначається як:

   $$cost[i][j] = min_{i \le r \le j}(cost[i][r-1] + cost[r+1][j] + \sum_{k=i}^{j} weight[k])$$

   де:
   - r - індекс кореня піддерева;
   - $cost[i][r-1]$ - вартість лівого піддерева (елементи до r);
   - $cost[r+1][j]$ - вартість правого піддерева (елементи після r);
   - $weight[k]$ - вага (ймовірність) елемента k.

3. **Обчислення кореня**: Для кожного піддерева обчислюється оптимальний корінь, який забезпечує мінімальну загальну вартість пошуку для всіх елементів цього піддерева. У масиві $root[i][j]$ зберігається індекс кореня для піддерева.
4. **Побудова дерева**: Після обчислення мінімальних вартостей і визначення коренів піддерев можна побудувати саме дерево. Починаємо з кореня і рекурсивно будуємо ліве та праве піддерева.
5. **Пошук у дереві**: Пошук у дереві реалізується за допомогою стандартного алгоритму бінарного пошуку, починаючи з кореня. Час виконання пошуку визначається глибиною вузла в дереві.
6. **Середня довжина пошуку**: Для обчислення середньої довжини пошуку використовується формула, що враховує ймовірності звернення до кожного елемента та їх глибину в дереві:

   $$Average\ search\ length\ = \frac{\sum_{i=1}^{n} depth(i) \times weight(i)}{n}$$

   де $depth(i)$ - глибина елемента i в дереві, а $weight(i)$ - його вага.

Складність алгоритму

Реалізація алгоритму фактично займає **O(n³)** часу. Пошук у побудованому дереві: **O(n)** у найгіршому випадку.

Мова реалізації алгоритму

C++

Модулі програми

- **`class Complex;`**

  Реалізує комплексне число з операціями +, ==, < та вивід числа.

- **`double magnitude()`**

  Додаткова функція для обчислення модуля комплексного числа (для сортування).

- **`struct KeyWeight`**

  Пара: комплексне число + вага.

- **`struct TreeNode`**

  Вузол дерева пошуку.

- **`TreeNode* createSubtree()`**

  Рекурсивна функція, яка будує піддерево з таблиці коренів.

- **`TreeNode* buildOptimalBST()`**

  Основна функція побудови оптимального бінарного дерева пошуку.

- **`void printTree()`**

  Виводить в консоль дерево в зручному форматі.

- **`bool search()`**

  Пошук елемента у дереві.

- **`int main()`**

  Головна функція програми, у якій:

  - створюється вектор keys з тестовими комплексними числами та відповідний вектор weights;
  - формується вектор keyWeights і сортується за модулем ключів;
  - виводиться відсортований список ключів із вагами;
  - будується оптимальне БДП;
  - виводиться структура побудованого дерева;
  - виконується пошук заданого значення в дереві;
  - обчислюється і виводиться середня довжина пошуку в дереві.

Інтерфейс користувача

Вхідні дані вводяться програмно (в функції int main()). Результат виводиться в консоль.

Тестові приклади

1) Вхідні дані:

Індекс	Ключ (K)	Модуль комплекс. числа	Вага (w)
0	1+0i	1.0	0.2
1	1+1i	$\sqrt{2} \approx 1.414$	0.3
2	0+2i	2.0	0.5

Ключі вже відсортовані за модулем:

K[0] = 1+0i w = 0.2

K[2] = 1+1i w = 0.3

K[3] = 0+2i w = 0.5

Позначення:

- cost[i][j] — мінімальна вартість дерева для підмножини ключів від i до j. Для обчислення ми перебираємо всі можливі корені r в діапазоні [i, …, j]:

  cost[i][j] = min (

  cost[i][r-1] + cost[r+1][j] + sum(i, j)

  для всіх r від i до j

)

- sum[i][j] — сума ваг для ключів від i до j - додається, бо кожне піддерево “падає” на рівень вниз -> його вартість зростає.

Обчислимо sum[i][j]:

i	j	sum
0	0	0.2
0	1	0.2+0.3=0.5
0	2	0.2+0.3+0.5=1.0
1	1	0.3
1	2	0.3+0.5=0.8
2	2	0.5

Звідси:

1. Довжина 1:

   cost[0][0] = 0.2; cost[1][1] = 0.3; cost[2][2] = 0.5.

2. Довжина 2:

   1) cost[0][1] - ключі [0,1]

      Можливі корені:

      - r = 0 (ліве піддерево -, вартість 0;
        праве піддерево: cost[1][1] = 0.3;
        сумарно 0.3 + sum[0][1] = 0.3 + 0.5 = 0.8)
      - r = 1 (ліве піддерево: cost[0][0] = 0.2;
        праве піддерево -, 0;
        сумарно 0.2 + 0.5 = 0.7)

      Отже, cost[0][1] = 0.7, корінь r = 1.

   2) cost[1][2] - ключі [1,2]

      Можливі корені:

      - r = 1 (ліве -, 0;
        праве: cost[2][2] = 0.5;

сумарно 0.5 + 0.8 = 1.3)

- r = 2 (ліве: cost[1][1] = 0.3;

  праве -, 0;

  сумарно 0.3 + 0.8 = 1.1)

Отже, cost[1][2] = 1.1, корінь r = 2.

3. Довжина 3:

cost[0][2] - ключі [0,1,2]

1) r = 0 (ліве -, праве cost[1][2] = 1.1;

   разом 1.1+1.0 = 2.1);

2) r = 1 (ліве cost[0][0] = 0.2, праве cost[2][2] = 0.5;

   разом 0.2+0.5+1.0 = 1.7);

3) r = 2 (ліве cost[0][1] = 0.7, праве -;

   разом 0.7+1.0 = 1.7);

Можливі корені: 1 або 2.

У висновку, один з можливих варіантів:

```
          1+1i
         /    \
    1+0i        0+2i
```

Тепер глянемо, що виводить програма, а також спробуємо знайти ключ, якого немає у дереві:

```
Keys (sorted by module):
Key 0: 1+0i, Weight: 0.2, |z| = 1
Key 1: 1+1i, Weight: 0.3, |z| = 1.41421
Key 2: 0+2i, Weight: 0.5, |z| = 2

Optimal binary search tree:
Key: 1+1i
weight 0.3
|       LEFT Key: 1+0i
|       LEFT weight 0.2
|       RIGHT Key: 0+2i
|       RIGHT weight 0.5

Finding the complex number 3+1i: Not found

Average search length: 1.7
```

2) Вхідні дані:

Індекс	Ключ (K[n-1])	Модуль комплекс. числа	Вага (w[n-1])
0	1+2i	$\sqrt{5} \approx 2.24$	0.1
1	-1+1i	$\sqrt{2} \approx 1.41$	0.2
2	3+0i	3.0	0.3
3	0-2i	2.0	0.15
4	-2-2i	$\sqrt{8} \approx 2.83$	0.25

Далі побудуємо оптимальне BST за допомогою програми, також перевіримо, чи знайде вона у дереві ключ, який в ньому є:

```
Keys (sorted by module):
Key 0: -1+1i, Weight: 0.2, |z| = 1.41421
Key 1: 0-2i, Weight: 0.15, |z| = 2
Key 2: 1+2i, Weight: 0.1, |z| = 2.23607
Key 3: -2-2i, Weight: 0.25, |z| = 2.82843
Key 4: 3+0i, Weight: 0.3, |z| = 3

Optimal binary search tree:
Key: -2-2i
weight 0.25
|       LEFT Key: 0-2i
|       LEFT weight 0.15
|       |       LEFT Key: -1+1i
|       |       LEFT weight 0.2
|       |       RIGHT Key: 1+2i
|       |       RIGHT weight 0.1
|       RIGHT Key: 3+0i
|       RIGHT weight 0.3

Finding the complex number 0-2i: Found

Average search length: 2.05
```

Висновки

Оптимальне бінарне дерево пошуку — це структура даних, яка дозволяє ефективно виконувати операції пошуку при відомій частоті доступу до ключів. Завдяки використанню динамічного програмування, можна побудувати дерево, що мінімізує середню кількість порівнянь під час пошуку.

Використані джерела

- Wikipedia
- Abdul Bari - Optimal Binary Search Tree (Successful Search Only) - Dynamic Programming (YouTube)
- Abdul Bari - Optimal Binary Search Tree Successful and Unsuccessful Probability - Dynamic Programming (YouTube)