THỰC NGHIỆM CÁC THUẬT TOÁN SẮP XẾP

PHẠM THẠCH THANH TRÚC

LỚP KHTN2022

CHUYÊN NGÀNH KHOA HỌC MÁY TÍNH

THÁNG 3 NĂM 2022

Thực nghiệm các thuật toán sắp xếp

Phạm Thạch Thanh Trúc

MSSV: 22521551

Lớp KHTN2022 – Khoá 17

Đại học Công Nghệ Thông Tin

Bản tóm tắt: Thuật toán sắp xếp là một trong những vấn đề cơ bản nhất của ngành công nghệ thông tin hiện nay. Việc tìm một thuật toán sắp xếp toàn năng đã luôn là một chủ đề sôi nổi nhất là đối với nửa thế kỉ trở lại đây. Hiện nay, lượng dữ liệu và thông tin cần qua xử lí là rất lớn, yêu cầu con người ta phải tìm cách nào đó sắp xếp các dữ liệu này một cách hiệu quả. Bài viết này sẽ tìm hiểu và phân tích qua một vài thuật toán bằng cách so sánh thời gian chạy của chúng.

---

1. Mở đầu:

Trong thời đại bây giờ, lượng dữ liệu cần lưu trữ là rất lớn, việc tìm ra một thuật toán sắp xếp tối ưu để có thể sắp xếp lượng dữ liệu ấy là một mối quan ngại không chỉ đối với các công ty lớn mà là một câu hỏi được đặt ra cho các nhà toán học và lập trình.

Có nhiều các thuật toán đồng thời là các cách tiếp cận khác nhau khi nói đến việc sắp xếp ví dụ như Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Selection Sort, Heap Sort. Tuy nhiên, Bài phân tích sau sẽ tập trung vào việc thử nghiệm của 4 thuật toán sắp xếp: Merge Sort, Quick Sort, Heap Sort và `std::sort` của c++ để có thể phân tích thời gian chạy và độ ổn định của chúng. Bài phân tích được thực hiện bằng cách cho chúng chạy lần lượt qua 1 triệu tham số khác nhau và sắp xếp lại.

Tất cả code về thuật toán sắp xếp, sinh test và các bộ test dược dùng đều được lưu ở Github: Link: [phamtruc-work/Analyzing-Sorting-Algorithm]

2. Các thuật toán sử dụng

2.1. Heap Sort

Heap Sort là một thuật toán sắp xếp dữ liệu bằng cách cho các phần tử vào một cây nhị phân hoàn chỉnh hay còn có tên gọi khác là heap (đống). Thuật toán này hoạt động bằng đưa dữ liệu vào thành một heap, sau đó lấy ra phần tử lớn nhất của heap và đưa nó vào cuối danh sách sắp xếp. Tiếp theo, thuật toán giảm kích thước của heap đi 1, khôi phục tính chất heap bằng cách sắp xếp lại các phần tử còn lại trong heap và lặp lại các bước trên cho đến khi toàn bộ dữ liệu được sắp xếp.

Thuật toán Heap Sort được sử dụng trong bài nghiên cứu.

```
void heapify(vector<double> &arr, int N, int i){
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < N && arr[l] > arr[largest])
        largest = l;
    if (r < N && arr[r] > arr[largest])
        largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, N, largest);
    }
}
void Heap Sort(vector<double> &arr, int N)
{
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);
    for (int i = N - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

2.2. Merge Sort

Merge sort sử dụng chia để trị (divide and conquer) để chia nhỏ dữ liệu thành các phần nhỏ hơn, sau đó sắp xếp từng phần đó và ghép lại để tạo ra một dãy đã sắp xếp.

Cụ thể, Merge Sort hoạt động bằng cách chia nhỏ một danh sách dữ liệu ban đầu thành các danh sách nhỏ hơn, tại mỗi cấp độ sẽ thực hiện việc ghép các danh sách đã sắp xếp lại với nhau, từ danh

sách nhỏ hơn đến danh sách lớn hơn, cho đến khi tất cả các phần tử được sắp xếp thành một danh sách duy nhất.

Thuật toán Merge Sort được sử dụng trong bài nghiên cứu:

```
void merge(vector<double> &arr, int const left, int const mid, int
const right){
    auto const tempArr1 = mid - left + 1;
    auto const tempArr2 = right - mid;
    auto *leftArr = new int[tempArr1], *rightArr = new int[tempArr2];
    for (auto i = 0; i < tempArr1; i++) leftArr[i] = arr[left + i];
    for (auto j = 0; j < tempArr2; j++) rightArr[j] = arr[mid + 1 +j];
    auto indexArr1 = 0, indexArr2 = 0;
    int indexMergedArr = left;
    while (indexArr1 < tempArr1 && indexArr2 < tempArr2){
        if (leftArr[indexArr1] <= rightArr[indexArr2]){
            arr[indexMergedArr] = leftArr[indexArr1];
            indexArr1++;
        } else{
            arr[indexMergedArr] = rightArr[indexArr2];
            indexArr2++;
        }
        indexMergedArr++;
    }

    while (indexArr1 < tempArr1){
        arr[indexMergedArr] = leftArr[indexArr1];
        indexArr1++; indexMergedArr++;
    }
    while (indexArr2 < tempArr2){
        arr[indexMergedArr] = rightArr[indexArr2];
        indexArr2++; indexMergedArr++;
    }
}

void Merge Sort(vector<double> &arr, int const low, int const high){
    if (low >= high)
        return;
    auto mid = low + (high - low) / 2;
    Merge Sort(arr, low, mid);
    Merge Sort(arr, mid + 1, high);
    merge(arr, low, mid, high);
}
```

2.3. Quick Sort

Quick sort sử dụng chia để trị (divide and conquer) để sắp xếp dữ liệu. Quick sort hoạt động bằng cách chọn một phần tử chốt (pivot) từ danh sách dữ liệu đầu vào và phân chia các phần tử khác thành hai danh sách con, một danh sách con chứa các phần tử nhỏ hơn phần tử chốt và một danh sách con chứa các phần tử lớn hơn phần tử chốt. Sau đó, thuật toán sử dụng đệ quy để sắp xếp hai danh sách con này. Quá trình chia nhỏ và sắp xếp được tiếp tục cho đến khi danh sách dữ liệu đầu vào được sắp xếp hoàn chỉnh. Để thuật toán Quick Sort được tối ưu hơn, bài thực nghiệm chọn phần tử chốt là ở giữa mảng nhằm các trường hợp xấu chạy quá lâu.

Thuật toán Quick Sort được sử dụng trong bài nghiên cứu:

```
int partition(vector<double>&arr, int start, int end)
{
    int pivot = arr[(start+end)/2];
    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }
    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);
    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {
        while (arr[i] <= pivot) {
            i++;
        }
        while (arr[j] > pivot) {
            j--;
        }
        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }
    return pivotIndex;
}

void Quick Sort(vector<double>&arr, int start, int end)
{
```

```
    if (start >= end)
        return;
    int p = partition(arr, start, end);
    Quick Sort(arr, start, p - 1);
    Quick Sort(arr, p + 1, end);
}
```

2.4. std::sort

std::sort là một thuật toán sắp xếp được cài đặt sẵn trong thư viện <algorithm> của c++. std::sort là một biến thể của Quick Sort có tên là Intro Sort, một thuật toán kết hợp tốc độ của Quick Sort và dùng Heap Sort trong những trường hợp xấu của Quick Sort.

Cách bài thực nghiệm dùng std::sort:

```
std::sort(arr.begin(),arr.end());
```

3. Tiến hành thực nghiệm:

3.1. Trước khi thực nghiệm:

Trước khi tiến hành thực nghiệm, ta hãy xem qua độ phức tạp bộ nhớ và thời gian của từng thuật toán sắp xếp để có cái nhìn rõ hơn.

	Tốt	Trung bình	Xấu	Bộ nhớ
Heap Sort	$nlogn$	$nlogn$	$nlogn$	1
Merge Sort	$nlogn$	$nlogn$	$nlogn$	n
Quick Sort	$nlogn$	$nlogn$	n^2	$logn$
std::sort	$nlogn$	$nlogn$	n^2	$logn$

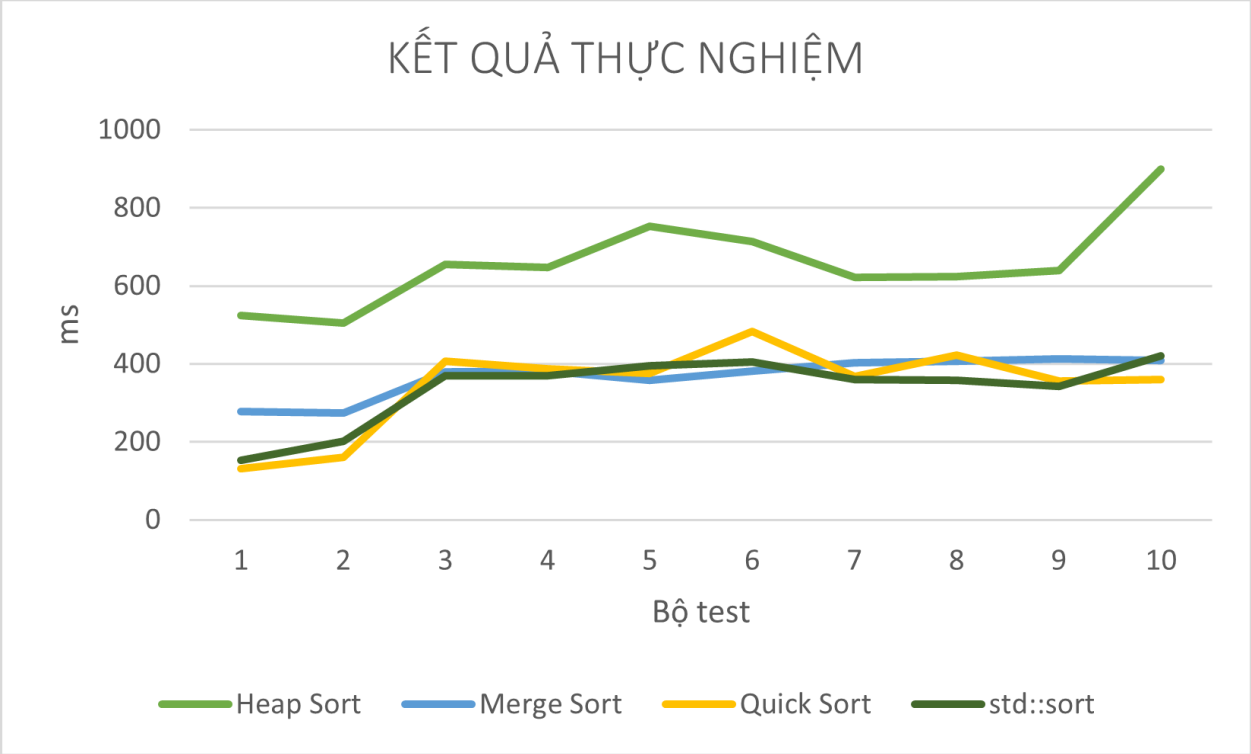
Tốc độ của từng thuật toán có thể bị ảnh hưởng và thay đổi theo từng thiết bị phần cứng. Thiết bị được sử dụng trong bài viết này gồm Intel i3-8100 3.60Ghz chạy trên hệ điều hành Window 10. Chương trình được thực thi bằng gcc 12.2.0.

Code sinh bộ test sử dụng thư viện random và dùng `std::mt19937` thay cho `rand()`. `std::mt19937` sinh ra các số ngẫu nhiên dựa trên một thuật toán phức tạp hơn và có một chu kỳ dài hơn, nên sẽ cho ra các giá trị ngẫu nhiên phân phối đồng đều hơn và giúp giảm khả năng lặp lại giá trị. Giá trị của 1 triệu số thực ấy nằm từ -10^9 đến 10^9 để giảm thiểu việc xuất hiện các số

trùng nhau và chương trình sinh ra bộ test ấy được viết bên ngoài để không ảnh hưởng tới thời gian thực thi chương trình của từng thuật toán sắp xếp. Bộ dữ liệu bao gồm 10 bộ test, mỗi bộ gồm 1 triệu số thực (ngẫu nhiên); bộ test thứ nhất đã có thứ tự tăng dần, bộ test thứ hai có thứ tự giảm dần, 8 dãy còn lại trật tự ngẫu nhiên.

Ngoài thời gian chạy ra, ta cũng cần phải để ý đến một số yếu tố như bộ nhớ sử dụng và độ ổn định của từng thuật toán để có một cách nhìn và đánh giá khách quan hơn.

3.2. Kết quả thực nghiệm:



Như có thể thấy, Heap Sort chạy chậm hơn nhiều so với các thuật toán sắp xếp còn lại tuy có cùng độ phức tạp trung bình là $nlogn$. Trung bình, Heap Sort có tốc độ chạy là 658ms trong khi các thuật toán còn lại ổn định ở mức 400ms. Merge Sort, Quick Sort và std::sort của c++ tuy trông giống nhau nhưng nếu chú ý thì sẽ thấy Merge Sort và std::sort có tốc độ chạy ổn định hơn so với Quick Sort. Điều này cũng không quá bất ngờ khi std::sort là Quick Sort nhưng đã được tối ưu hoá lên. Kết luận, trong khi Heap Sort.

3.3. Đánh giá thực nghiệm:

- Ưu điểm của từng thuật toán:

Heap Sort: Không tốn nhiều bộ nhớ, các trường hợp chạy ở $O(nlogn)$

Merge Sort: Thuật toán có độ ổn định cao cùng tốc độ chạy nhanh là $O(nlogn)$.

Quick Sort: Thuật toán chạy nhanh và ổn định nếu được cài đặt tốt. Tốc độ chạy là $O(nlogn)$.

std::sort: Sự kết hợp hoàn hảo của việc chạy nhanh và ổn định. Là thuật toán tốt nhất nếu nhìn chung.

- Nhược điểm của từng thuật toán:

Heap Sort: Không có sự ổn định.

Merge Sort: Cài đặt phức tạp, cần nhiều bộ nhớ để lưu các mảng con.

Quick Sort: Trường hợp xấu nhất là $O(n^2)$ và không ổn định trong một số trường hợp.

std::sort: Trường hợp xấu nhất là $O(n^2)$ nhưng rất khó để tìm các trường hợp ấy.

4. Tổng kết

Nhìn chung, cả 4 thuật toán đều nhanh và hiệu quả trong việc sắp xếp các số. Riêng Heap Sort, tuy thuật toán có tốc độ lâu hơn các thuật toán khác, khi được dùng chung với một thuật toán khác như Quick Sort như ta thấy ở std::sort, thì sẽ giúp khắc phục được việc không ổn định của Quick Sort.

Vậy, nhìn chung thì std::sort đã chứng minh được sự tối ưu khùng khiếp và sức mạnh của Introsort. Bằng việc kết hợp nhiều thuật toán sắp xếp lại với nhau (Quick Sort, Insertion Sort và Heap Sort), std::sort của c++ đã cho ra kết quả tốt nhất với việc tốc độ thực thi nhanh cùng độ ổn định tốt.