

Chặt nhị phân song song

Chặt nhị phân song song

Tác giả:

- Cao Thanh Hậu - Trường Đại học Khoa học Tự Nhiên, ĐHQG-HCM

Reviewer:

- Ngô Nhật Quang - University of Texas at Dallas
- Nguyễn Minh Nhật - THPT chuyên Khoa học Tự nhiên, ĐHQGHN
- Nguyễn Minh Hiến - Đại học Công nghệ, ĐHQGHN
- Nguyễn Đức Kiên, Trường Đại học Công nghệ, ĐHQGHN

Giới thiệu

Chặt nhị phân đã là một cụm từ vô cùng quen thuộc với các bạn học lập trình, là một thuật toán hữu dụng, dễ dùng và dễ dàng kết hợp với nhiều thuật toán khác.

Ở bài viết này, chúng ta cùng tìm hiểu cách để kết hợp thuật toán chặt nhị phân với "chính nó" - chặt nhị phân song song.

Khác với chặt nhị phân, chặt nhị phân song song là khái niệm có phần ít phổ biến hơn. Đúng như tên gọi của nó, cốt lõi của thuật toán chặt nhị phân song song là chặt nhị phân, vì vậy bạn đọc nên có trước các kiến thức cơ bản về chặt nhị phân để tiện hiểu rõ bài viết này.

Bài toán

Meteors

Có N tổ chức thành viên cùng tham gia thu thập các mảnh đá vũ trụ trên quỹ đạo tròn, thành viên thứ i cần thu được ít nhất p_i mảnh đá. Quỹ đạo tròn được chia thành M khu vực, trong đó khu vực M và khu vực 1 nằm cạnh nhau, mỗi khu vực thuộc về một trong n tổ chức thành viên.

Các nhà khoa học đã dự đoán được sự xuất hiện của K cơn mưa sao băng, cơn mưa xuất hiện vào thời điểm i sẽ giúp các khu vực có chỉ số thuộc đoạn $[l_i; r_i]$ tăng thêm a_i mảnh đá (nếu $l_i > r_i$ thì đoạn $[l_i; r_i]$ được chia thành 2 đoạn $[l_i; m]$ và $[1; r_i]$).

Với mỗi tổ chức thành viên, cho biết thời điểm sớm nhất mà tổ chức đó thu thập đủ số mảnh yêu cầu.

$$1 \leq N, M, K \leq 3 \times 10^5, 1 \leq o_i, l_i, r_i \leq N, 1 \leq g_i \leq 10^9, 1 \leq x_i \leq 10^9$$

Ngây thơ

Với mỗi tổ chức, thực hiện chặt nhị phân để tìm thời điểm sớm nhất tổ chức đó thu thập đủ số mảnh.

Để kiểm tra xem đến thời điểm t thì tổ chức i đã thu thập được bao nhiêu mảnh, ta duyệt qua các cơn mưa sao băng có thời điểm xuất hiện không quá t , dùng cấu trúc dữ liệu Fenwick Tree (cây BIT) để cập nhật các thông tin và cuối cùng lấy tổng các mảnh đá tại những vị trí mà i sở hữu trên quỹ đạo.

Với cách làm trên, độ phức tạp để tìm đáp án cho tổ chức i là $O((K + C_i) \times \log(K) \times \log(M))$, trong đó C_i là số vị trí mà i sở hữu. Tổng độ phức tạp là $O((N \times K + M) \times \log(K) \times \log(M))$, tất nhiên là quá giới hạn thời gian.

```

1  #define For(i, a, b) for (int i = a; i <= b; ++i)
2
3  int n, m, k;
4  int owner[N], req[N], L[N], R[N], A[N], l[N], r[N];
5  vector<int> pos[N];
6  BIT bit;
7
8  void update_info(int i, int d) {
9      if (L[i] <= R[i]) {
10         bit.add(L[i], R[i], d * A[i]);
11     }
12     else {
13         bit.add(L[i], m, d * A[i]);
14         bit.add(1, R[i], d * A[i]);
15     }
16 }
17
18 bool check(int o, int k) {
19     For(i, 1, k) {
20         update_info(i, 1);
21     }
22     long long cnt = 0;
23     for (int i : pos[o]) {
24         cnt += bit.get(i);
25         cnt = min(cnt, (long long)req[o] + 1);
26     }
27     For(i, 1, k) {
28         update_info(i, -1);
29     }
30     return cnt >= req[o];
31 }
32
33 int main() {
34     cin >> n >> m;
35     For(i, 1, m) cin >> owner[i];
36     For(i, 1, n) cin >> req[i];
37
38     cin >> k;
39     For(i, 1, k) cin >> L[i] >> R[i] >> A[i];
40

```

```

41     For(i, 1, n) {
42         l[i] = 0, r[i] = k + 1;
43     }
44     For(i, 1, m) {
45         pos[owner[i]].push_back(i);
46     }
47
48     bit.makeBIT(m);
49     For(i, 1, n) {
50         int l = 0, r = k + 1;
51         while (l < r) {
52             int g = (l + r) / 2;
53             if (check(i, g)) r = g;
54             else l = g + 1;
55         }
56         if (r > k) cout << "NIE\n";
57         else cout << r << "\n";
58     }
59
60     return 0;
61 }

```

Tối ưu

Gọi $check(t, k)$ là hàm kiểm tra liệu k cơn mưa đầu tiên có đáp ứng đủ số mảnh đá yêu cầu của tổ chức t hay không.

Giả sử ta gọi hàm $check(t, K)$, cần xét qua tất cả K thông tin nhưng lại chỉ dùng để tìm kết quả cho một hàm $check$, rất phí thời gian. Vậy, liệu có thể dừng lại ở thông tin thứ i để tiện thể trả lời hàm $check(t', i)$ cho một tổ chức nào khác không?

Tiếp tục ý tưởng trên, gọi $l[i], r[i]$ là đoạn chứa đáp án cho tổ chức thứ i .

Lần lượt xét qua từng thông tin về các cơn mưa và cập nhật lên cây BIT, tại thông tin thứ k , đồng thời gọi hàm $check(t, k)$ với t là những tổ chức mà $(l[t] + r[t])/2 = k$. Vậy từ $l[t], r[t]$ và $check(t, k)$ ta có thể cập nhật lại $l[t]$ và $r[t]$ giống như chặt nhị phân.

Tính toán lại, với mỗi lần xét qua K thông tin như vậy, ta có:

- Độ phức tạp cho việc cập nhật các thông tin là $K \times \log(M)$.
- Độ phức tạp cho việc gọi hàm $check$ là $M \times \log(M)$ (vì mỗi vị trí trên quỹ đạo được xét qua nhiều nhất 1 lần).
- Với mọi tổ chức t , độ dài đoạn $l[t], r[t]$ được giảm đi một nửa.

Vậy chỉ cần lặp lại việc xét duyệt như trên đến khi đoạn $l[t], r[t]$ của mọi tổ chức t có độ dài 1. Cần nhiều nhất $\log(K)$ lần xét, độ phức tạp là $O((M + K) \times \log(M) \times \log(K))$.

Cài đặt

```

1  int owner[N], req[N], L[N], R[N], A[N], l[N], r[N];
2  vector<int> pos[N];

```

```

3   BIT bit;
4
5   bool check(int o) {
6       long long cnt = 0;
7       for (int i : pos[o]) {
8           cnt += bit.get(i);
9           cnt = min(cnt, (long long)req[o] + 1);
10      }
11      return cnt >= req[o];
12  }
13
14  int main() {
15      int n, m; cin >> n >> m;
16      For(i, 1, m) cin >> owner[i];
17      For(i, 1, n) cin >> req[i];
18
19      int k; cin >> k;
20      For(i, 1, k) {
21          cin >> L[i] >> R[i] >> A[i];
22      }
23
24      For(i, 1, n) {
25          l[i] = 0, r[i] = k + 1;
26      }
27      For(i, 1, m) {
28          pos[owner[i]].push_back(i);
29      }
30
31      int processing = 1;
32      vector<vector<int>> queries;
33      while (processing) {
34          processing = 0;
35
36          queries.assign(k + 5, vector<int>(0));
37          bit.makeBIT(m);
38
39          For(o, 1, n) {
40              if (l[o] >= r[o]) continue;
41              processing = 1;
42              queries[(l[o] + r[o]) / 2].push_back(o);
43          }
44
45          For(ki, 0, k) {
46              if (ki) {
47                  if (L[ki] <= R[ki]) {
48                      bit.add(L[ki], R[ki], A[ki]);
49                  }
50                  else {
51                      bit.add(L[ki], m, A[ki]);
52                      bit.add(1, R[ki], A[ki]);
53                  }
54          }

```

```

55         }
56         for (int o : queries[ki]) {
57             if (check(o)) r[o] = ki;
58             else l[o] = ki + 1;
59         }
60     }
61     queries.clear();
62     bit.clear();
63 }
64
65 For(o, 1, n) {
66     if (r[o] <= k) cout << r[o] << "\n";
67     else cout << "NIE\n";
68 }
69
70 return 0;
71 }

```

Trong cài đặt trên, hàm $check(o, k)$ được thay bằng $check(o)$, vì đã có cây BIT "chung" chứa thông tin của k cơn mưa đầu tiên.

Lưu ý, việc xóa dữ liệu của *bit* và *queries* sau mỗi lần chặt đảm bảo độ phức tạp không gian cho thuật toán là $O(N + M + K)$, nếu không xóa sẽ dẫn đến $O((M + K) \times \log(K))$.

Một cách cài đặt khác sử dụng đệ quy như sau:

Giả sử cần tìm đáp án cho các thành viên thuộc tập s , được đảm bảo rằng các đáp án đều nằm trong đoạn $[l; r]$, ta có thể xây dựng hàm $solve(s, l, r)$ như bên dưới:

```

1  p = 0;
2  update(t):
3      while (p < t) update_info(++p, +1);
4      while (p > t) update_info(p--, -1);
5
6  solve(s, l, r):
7      if (l == r):
8          for each i in s: ans[i] = l;
9          return;
10
11     mid = (l + r) / 2;
12     update(mid);
13     for each i in s:
14         if (count(i) >= require[i]):
15             add i to left_set;
16         else:
17             add i to right_set;
18     s.clear();
19
20

```

```
solve(left_set, l, mid);
solve(right_set, mid + 1, r);
```

Trong đó, hàm $count(i)$ đếm số mảnh đá tổ chức i thu được, lấy từ cây BIT được tạo ở hàm $update$, $require[i]$ là số mảnh đá tổ chức i cần thu.

Hàm $update$ được cài đặt khá đặc biệt, giá trị p cho biết cây BIT đang lưu thông tin các cơn mưa sao băng có thời điểm xuất hiện không quá p . Khi $update(t)$ được gọi, di chuyển p đến t và cập nhật các thông tin được của các cơn mưa được duyệt qua.

Theo cách này, đối với các bài mà việc "đi ngược" trong hàm $update$ không thực hiện được, ta cần phân cấp các hàm $solve()$: hàm $solve()$ được gọi đầu tiên có cấp 1, các hàm được gọi từ hàm cấp i thì thuộc cấp $i + 1$. Như vậy các hàm $update$ cho cùng một cấp sẽ không cần đi ngược.

Tổng quát

Bài toán trên có thể chưa đầy đủ để đưa ra một cấu trúc chung cho thuật toán, và trên thực tế có nhiều trường hợp mà thuật toán hoạt động "trá hình", nghĩa là sử dụng các tính chất được phân tích bên trên nhưng theo một cách áp dụng rất khác, khó để đưa ra bất kỳ quy chuẩn nào.

Tuy nhiên, để dễ nhận diện bài toán trong nhiều trường hợp, ta có thể tạm hiểu một cách tổng quát thuật toán chặt nhị phân song song như sau:

Trường hợp áp dụng

Chặt nhị phân song song là cải tiến cho việc thực hiện nhiều lần chặt nhị phân tương tự nhau. Vì vậy bài toán thường liên quan đến các truy vấn giống nhau, và mỗi truy vấn có thể được giải quyết bằng chặt nhị phân. Bên cạnh đó, hàm kiểm tra trong chặt nhị phân thường cần xét qua một tiền tố của một dãy thông tin cho trước.

Một ví dụ điển hình là bài toán tìm thời điểm đầu tiên mà mỗi điều kiện trong dãy điều kiện cho trước được thỏa mãn.

Thuật toán

Ví dụ với dạng bài toán: Cho Q cập nhật được thực hiện lần lượt theo thứ tự và K điều kiện. Với mỗi điều kiện, cho biết thời điểm sớm nhất nó được thỏa mãn (giả sử đề bài đảm bảo sau thời điểm sớm nhất đó thì điều kiện luôn được thỏa mãn).

Để cài đặt thuật toán, ta cần chuẩn bị:

- Cấu trúc dữ liệu để thực hiện các cập nhật.
- Hai mảng L, R thể hiện đoạn chứa kết quả ứng với từng điều kiện.
- Mảng $check$ để lưu các điều kiện theo giá trị mid

Thuật toán như sau:

```
1 | Lặp lại log(Q) lần:
2 |     reset cấu trúc dữ liệu
3 |     reset mảng check
4 |
5 |     với mỗi cặp l[i], r[i] mà l[i] < r[i]:
```


```

6      mid = (l[i] + r[i]) / 2
7      thêm i vào check[mid]
8
9      với mỗi cập nhật thứ q:
10     thực hiện cập nhật thứ q vào cấu trúc dữ liệu
11     với mỗi giá trị i thuộc check[q]:
12         // thực hiện kiểm tra trên cấu trúc dữ liệu
13         nếu điều kiện thứ i được thỏa mãn:
14             r[i] = q
15         nếu không:
16             l[i] = q + 1

```

Trong phần nhiều các trường hợp, dãy điều kiện không được cho cụ thể trong đề bài, thường phải biến đổi bài toán thành các bài toán con và dùng chặt nhị phân song song để giải quyết các bài toán con đó.

Ví dụ

Thử áp dụng vào bài toán [Kết nối chơi game](#) .

Trong bài toán trên, có thể xem điều kiện thứ i trong M điều kiện là: tất cả các học sinh thích trò chơi thứ i được kết nối với nhau.

Cập nhật thứ i trong Q cập nhật là: kết nối máy tính của học sinh thứ $U[i]$ và $V[i]$.

Vậy là bài toán đã có dạng: với mỗi điều kiện, tìm thời điểm sớm nhất nó được thỏa mãn. Thuật toán y hệt như phần trước.

Luyện tập

[SRM 675 Div1 500](#) 

[LimitedMemorySeries1](#) 

[Travel in HackerLand](#) 

[Kết nối chơi game](#) 

Được cung cấp bởi [Wiki.js](#)