PROCESSAMENTO EM BATCHES COM DECOMPOSIÇÃO EM RAIZ QUADRADA

Paulo Correia, UFPE

Resumo

Este artigo apresenta uma estratégia que troca dinamismo por reconstruções periódicas: em vez de atualizar a estrutura principal a cada operação, acumula-se um conjunto de modificações e processa-se esse conjunto em lote. Foi motivado pelo editorial do problema Xenia and Tree no Codeforces, um dos exemplos mais conhecidos de aplicação dessa técnica [2].

1. Um problema introdutório

Dado um *array* a de tamanho N, responda a Q consultas dos seguintes tipos:

1. $l\ r\ x$: atualize $a_i \leftarrow a_i + x$ para todo $i \in [l, r]$
2. i: imprima o valor de a_i.

Embora esse problema admita uma solução clássica com *segment tree*, ele será utilizado para introduzir uma abordagem alternativa.

2. Solução geral

As operações de atualização serão armazenadas em um vetor, denominado *batch*, sem que as atualizações sejam imediatamente aplicadas ao *array*.

Ao processar uma consulta do tipo 2 na posição i, todas as atualizações presentes no *batch* serão percorridas, computando-se a contribuição de cada uma delas para a posição consultada.

Essa abordagem já é correta. No entanto, no pior caso, cada consulta pode exigir a inspeção de todas as atualizações pendentes, resultando em uma complexidade de $O(Q^2)$.

```cpp
vector<Query> batch;
GlobalStructure global;

for (Query query : queries) {
    if (query.isUpdate) {
        batch.push_back(query);
    } else {
        auto ans = global.get(query);

        for (Query update : batch)
            ans += contribution(update, query);

        cout << ans;
    }

    if (batch.size() > T) {
        global.apply(batch);
        batch.clear();
    }
}
```

CÓDIGO 1: SOLUÇÃO GERAL.

Limitando o *batch* a no máximo T atualizações e denotando por $C_{\text{contribution}}$ o custo de calcular a contribuição de uma atualização para uma consulta, cada consulta do tipo 2 possui custo $O(T \cdot C_{\text{contribution}})$. Ao atingir o limite T, o *batch* é aplicado de uma vez. Como essa operação ocorre a cada T atualizações, ela é realizada $O(Q/T)$ vezes. Denotando por C_{apply} o custo de aplicar todas as atualizações presentes no *batch*, obtém-se a seguinte complexidade total de $O(Q \cdot T \cdot C_{\text{contribution}} + C_{\text{apply}} \cdot (Q/T) + N)$, onde o termo N corresponde ao custo linear de inicialização da estrutura *global*.

3. Solução 1

Voltando ao problema introduzido, precisamos definir como calcular a contribuição de uma operação do tipo 1 numa posição i, qual estrutura usar, e como aplicar o *batch* nessa estrutura. Para calcular a contribuição de uma operação do tipo 1 na posição i, basta simplesmente checar se $l \le i \le r$.

Como estrutura global, utilizaremos um vetor global. Para aplicar o *batch*, pode-se usar uma técnica conhecida como *delta encoding* [4].

```cpp
void apply(vector<Query> batch) {
    vector<long long> delta(n + 1, 0);

    for (Query query : batch) {
        delta[query.l] += query.x;
        delta[query.r + 1] -= query.x;
    }

    long long prefix = 0;
    for (int i = 0; i < n; i++) {
        prefix += delta[i];
        global[i] += prefix;
    }
}
```

CÓDIGO 2: APLICAÇÃO DO BATCH UTILIZANDO DELTA ENCODING.

Complexidade temporal: $O(Q \cdot T + (N + T) \cdot (Q/T) + N)$. Reorganizando os termos, temos $O(Q \cdot (T + N/T) + Q + N)$. Como a função $T + N/T$ é minimizada para $T = \Theta(\sqrt{N})$, obtemos uma complexidade final de $O(Q \cdot \sqrt{N} + N + Q)$.

4. 2015 CodeCraft IIIT Hyderabad Replay - A

Nesta seção apresentamos a solução para o problema *Queries on the Tree* [3]. Nesse problema, considera-se uma árvore de N nós, enraizada no vértice 1. Inicialmente, cada nó armazena zero moedas. O objetivo é processar Q consultas dos seguintes tipos:

1. $d\ x$: adicione x moedas a todos os nós que estão a uma distância d da raiz.
2. u: imprima a soma das moedas dos nós da subárvore de u.

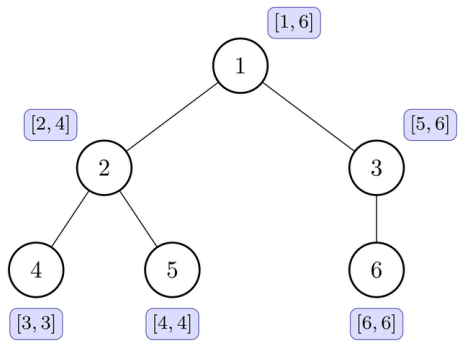
Utilizando a mesma técnica, guardaremos as operações do tipo 1 no *batch*. Usando um vetor *global* que armazena para cada nó a quantidade de moedas na sua subárvore após as aplicações

anteriores, podemos aplicar o *batch* em duas etapas: (i) acumulamos as atualizações pendentes por profundidade em um vetor *add* e (ii) percorremos a árvore com uma DFS, computando a contribuição de cada subárvore e atualizando global.

```cpp
using ll = long long;
vector<ll> add;

// (i)
void apply(vector<Update> batch) {
    add.assign(n + 1, 0);

    for (Update upd : batch)
        add[upd.d] += upd.x;

    dfs(root, root);
}

// (ii)
ll dfs(int u, int p, int d = 0) {
    ll subtree = add[d];

    for (int v : g[u]) {
        if (v == p) continue;
        subtree += dfs(v, u, d + 1);
    }

    global[u] += subtree;
    return subtree;
}
```

CÓDIGO 3: APLICAÇÃO DAS ATUALIZAÇÕES PENDENTES AO VETOR GLOBAL.

Como as consultas do tipo 2 pedem informações sobre subárvores, é conveniente linearizar a árvore de forma que cada subárvore corresponda a um intervalo contíguo. Para alcançar esse objetivo, realizamos uma *Euler tour* [1] da árvore. A travessia consiste em executar uma DFS mantendo uma variável de tempo. Ao visitar um vértice u pela primeira vez, armazenamos o instante atual em $tin[u]$ e incrementamos o tempo. Após finalizar o processamento de toda a subárvore de u, registramos o instante correspondente em $tout[u]$. Como consequência, u pertence à subárvore de v se, e somente se, $tin[v] \le tin[u] \le tout[v]$.



Vetor linearizado pelo Euler Tour:

Índice (Tempo)	1	2	3	4	5	6
Vértice (u)	1	2	4	5	3	6
$tin[u]$	1	2	3	4	5	6
$tout[u]$	6	4	3	4	6	6

FIGURA 1: VISUALIZAÇÃO DA EULER TOUR. CADA NÓ u APRESENTA SEUS TEMPOS NA FORMA [tin[u], tout[u]].

Agora, para responder a consulta, percorremos o *batch* e calculamos a contribuição de cada atualização (d_i, x_i) para o nó u. Isso se resume a contar quantos nós da subárvore de u estão em uma distância d_i da raiz.

Durante o *Euler tour*, além de armazenar os tempos de entrada e saída de cada vértice, mantemos também, para cada profundidade d, um vetor lvl contendo os valores de $tin[v]$ dos vértices nessa profundidade. Assim, basta contar quantos nós em $lvl[d]$ estão na subárvore de u. Como $lvl[d]$ foi inserido ordenadamente no *Euler tour*, podemos utilizar duas buscas binárias para encontrar quantos elementos pertencem ao intervalo $[tin[u], tout[u]]$, que corresponde a subárvore de u.

```cpp
ll subtree = global[u];

for (Update upd : batch) {
    auto lo = get_lower_bound(lvl[upd.d], tin[u]);
    auto hi = get_upper_bound(lvl[upd.d], tout[u]);

    subtree += upd.x * (hi - lo);
}

cout << subtree << "\n";
```

CÓDIGO 4: CÁLCULO DA CONTRIBUIÇÃO DAS ATUALIZAÇÕES PARA UMA CONSULTA DO TIPO 2.

A construção inicial da *Euler tour* e dos vetores auxiliares lvl e global possui custo $O(N)$. Para uma consulta do tipo 2, percorremos o *batch* e, para cada atualização, realizamos duas buscas binárias em $lvl[d]$, logo $C_{\text{contribution}} = O(\log N)$. Já a aplicação do *batch* consiste em acumular as atualizações por profundidade e executar uma DFS na árvore, resultando em $C_{\text{apply}} = O(N + T)$. Substituindo esses valores na complexidade geral da técnica, obtemos $O(N + Q \cdot T \log N + (Q/T) \cdot (N + T))$. Simplificando, temos $O(N + Q \cdot (T \log N + N/T))$. Uma escolha simples para o parâmetro é $T = \Theta(\sqrt{N})$, obtendo uma complexidade final de $O(N + Q \cdot \sqrt{N} \log N)$.

5. Considerações finais

O processamento em *batches* é uma técnica simples, mas bastante útil quando atualizações individuais são complexas de aplicar e consultas podem absorver um pequeno custo adicional. Ao agrupar operações e processá-las periodicamente, obtemos uma solução mais direta, sem recorrer a estruturas de dados mais sofisticadas.

Referências

[1] Benjamin Qi, Andrew Wang, Neo Wang. Euler Tour Technique. Disponível em: https://usaco.guide/gold/tree-euler. Acesso em: 21 jul. 2026.
[2] Pavel Kholkin. Codeforces Round #199 (Div. 2) Editorial. Disponível em: https://codeforces.com/blog/entry/8800. Acesso em: 13 jun. 2026.
[3] 2015 CodeCraft IIIT Hyderabad Replay. Queries on the Tree. Disponível em: https://codeforces.com/gym/100589/problem/A. Acesso em: 21 jul. 2026.
[4] Wikipedia. Delta Encoding. Disponível em: https://en.wikipedia.org/wiki/Delta_encoding. Acesso em: 26 jul. 2026.