V Escola de Inverno da Maratona de Programação

Congresso da Sociedade Brasileira de Computação 2024

Alberto Tavares Duarte Neto
Daniel Saad Nogueira Nunes
Edson Alves da Costa Júnior

24 de julho de 2024

Alberto Neto

Daniel Saad

- Professor do Instituto Federal de Brasília, *campus* Taguatinga.
- Técnico das equipes do IFB.
- Participou do comitê de sistemas das Maratonas de Programação da SBC.
- Doutor em Informática pela Universidade de Brasília.
- Tema: compressão e estruturas de dados compactadas para textos altamente repetitivos.
- E-mail: daniel.nunes@ifb.edu.br

Edson Alves

- Professor da Faculdade UnB Gama.
- Doutor em Engenharia Elétrica pela Universidade de Brasília.
- Tema: propagação de incertezas em eletromagnetismo.
- E-mail: edsonaves@unb.br

Definição (Passeio)

Um passeio de tamanho m é uma sequência de vértices $v_1, \ldots, v_m$ de modo que $(v_i, v_{i+1}) \in E, \quad 1 \leq i < n$.

Note que, de acordo com essa definição, podemos passar pelo mesmo vértice ou aresta mais de uma vez.

Alazão

Modelagem do problema

O problema quer saber, para um grafo $G = (V, E)$, se existe um passeio de tamanho múltiplo de k partindo do vértice de origem v_1 e chegando ao vértice destino v_n.

Alazão: solução

- Iremos resolver esse problema utilizando busca em profundidade com *memoization*.
- Armazenaremos em uma tabela $T[u][l] = 1$ se existe um caminho partindo da origem e chegando em u com tamanho de passeio $l \mod k$. Caso contrário, $T[u][l] = 0$.
- Se existe uma aresta (u, v) **e** é possível chegar ao vértice u com um passeio de tamanho $l \mod k$, então é possível chegar ao vértice v com um passeio de tamanho $l + 1 \mod k$.
- Fácil de implementar com uma busca em profundidade (DFS).
- Obviamente existe um passeio de tamanho múltiplo de k do vértice origem v_1 ao vértice destino v_n se $T[n][0] = 1$.

Alazão: solução

```
function<void(int,int)> dfs = [&](int u, int x) {
    vis[u][x] = 1;
    int y = (x+1)%k;
    for(int v=0;v<n;v++) {
        if(not adj[u][v]) continue;
        if(not vis[v][y]) dfs(v,y);
    }
};
dfs(0,0);
if(vis[n-1][0]) {
    cout << "Sim" << endl;
}
else {
    cout << "Nao" << endl;
}
```

Alazão

Análise de pior-caso

Como a busca em profundidade pode passar por cada nó v até k vezes, a complexidade da solução é $\Theta((|V| + |E|) \cdot k)$.

Estrutural

Modelagem do problema

Trata-se de descobrir um ciclo Hamiltoniano, isto é, um ciclo que começa em um vértice, passa por todos os outros, sem repeti-los, e retorna ao vértice original.

Observação

Em um ciclo Hamiltoniano, tanto faz o vértice de origem, já que é possível passar por todos os outros e retornar ao vértice original.

Estrutural: solução

Construção do grafo

Primeiramente, devemos construir um grafo não-direcionado.
Para cada par de strings (S_u, S_v), cria-se uma aresta (u, v) sempre que a **distância de Hamming** entre as strings for menor ou igual a X.
Esse processo leva tempo $\Theta(V^2)$.

Estrutural: solução

```
int hamming_distance(const string &s1, const string &s2) {
    assert(s1.size() == s2.size());
    int d = 0;
    for (size_t i = 0; i < s1.size(); i++) {
        if (s1[i] != s2[i])
            d++;
    }
    return d;
}
```

Estrutural: solução

```
vector<vector<bool>> build_graph(vector<string> &vs, int make_edge_threshold) {
    vector<vector<bool>> graph(n, vector<bool>(n, false));
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (hamming_distance(vs[i], vs[j]) <= make_edge_threshold) {
                graph[i][j] = true;
                graph[j][i] = true;
            }
        }
    }
    return graph;
}
```

Estrutural: solução

Caso base

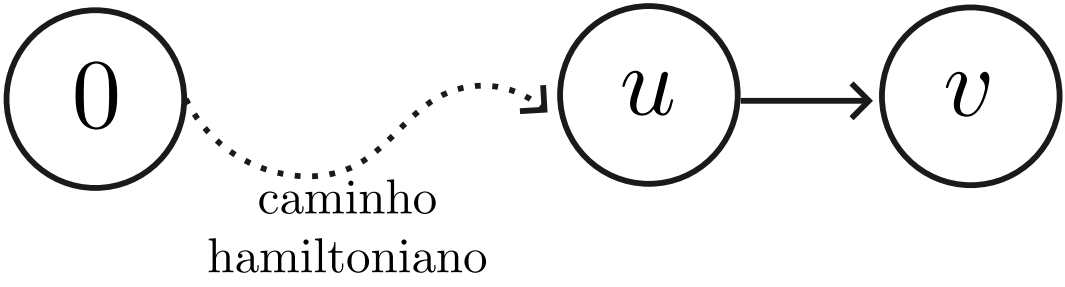
- O caso com $|V| = 1$ é fácil, basta imprimir o ciclo contendo o próprio vértice.
- Com $|V| > 1$, a solução pode ser obtida via **programação dinâmica**.

Estrutural: solução

Programação dinâmica

- Uma vez que o grafo $G = (V, E)$ foi construído, iremos resolver o problema por programação dinâmica.
- Suponha que para um subconjunto $S \subseteq V$, tenhamos um caminho hamiltoniano iniciando no vértice 0 e terminando no vértice u.
- Se existe uma aresta (u, v) e $v \notin S$, então existe um caminho hamiltoniano para o conjunto $S \cup \{v\}$ terminando em v.
- Além de armazenar se existe ou não o caminho hamiltoniano, guardaremos a informação do vértice predecessor, u, para recuperar o caminho hamiltoniano.

Estrutural: solução



Estrutural: solução

Programação dinâmica

$$T(v, S) = \begin{cases} -1, & v = 0 \text{ e } S = \{0\} \\ \text{u}, & u \in S \text{ e } T(S - \{v\}, u) \neq \perp \text{ e } u \neq v \text{ e } (u, v) \in E \\ \perp, & \text{caso contrário} \end{cases}$$

Observação

Existirá um ciclo hamiltoniano sempre que existir alguma entrada $T(v, V) \neq \perp$ e aresta $(v, 0) \in E$.

Estrutural: solução

```
vector<vector<int>> dp(vector<vector<bool>> &graph) {
    vector<vector<int>> matrix(n, vector<int>(1 << n, -2));
    matrix[0][1 << 0] = -1;
    for (int mask = 0; mask < 1 << n; mask++) {
        for (int j = 0; j < n; j++) {
            if ((mask & (1 << j)) == 0)
                continue;
            for (int i = 0; i < n; i++) {
                if (mask & 1 << i and i != j and
                    matrix[i][mask ^ (1 << j)] != -2 and graph[i][j]) {
                    matrix[j][mask] = i;
                }
            }
        }
    }
    return matrix;
}
```

Estrutural: solução

```
void solve() {
    auto graph = build_graph(addresses, x);
    auto matrix = dp(graph);
    bool has_sol = false;
    for (int i = 0; i < n; i++) {
        if (matrix[i][(1 << n) - 1] != -2 and graph[i][0]) {
            print_solution(matrix, addresses, i);
            has_sol = true;
            break;
        }
    }
    if (!has_sol) {
        cout << "impossivel\n";
    }
}
```

Estrutural: solução

```
void print_solution(const vector<vector<int>> &matrix,
                    const vector<string> &addresses, const int end_vertex)
    vector<string> path;
    int next_v = end_vertex;
    int mask = (1 << n) - 1;
    do {
        int cur_v = next_v;
        path.push_back(addresses[cur_v]);
        next_v = matrix[cur_v][mask];
        mask = mask ^ (1 << cur_v);
    } while (next_v != -1);
    reverse(path.begin(), path.end());
    for (int i = 0; i < n; i++) {
        cout << path[i] << " -> ";
    }
    cout << addresses[0] << "\n";
}
```

Estrutural: análise

Análise de pior-caso

O tempo de pior caso da solução é de $\Theta(2^{|V|} \cdot |V|^2)$, visto que a tabela de programação dinâmica tem $\Theta(2^{|V|} \cdot |V|)$ estados e, para computar cada um, gasta-se tempo $\Theta(|V|)$ no pior caso.

Fila da Padaria

Modelagem do problema

Dada uma sequência de elementos $v_1, \ldots, v_n$, o problema quer saber, para cada v_i, a quantidade de elementos v_j, $j < i$, isto é, à esquerda de v_i, que são maiores que v_i.

Fila da padaria: solução

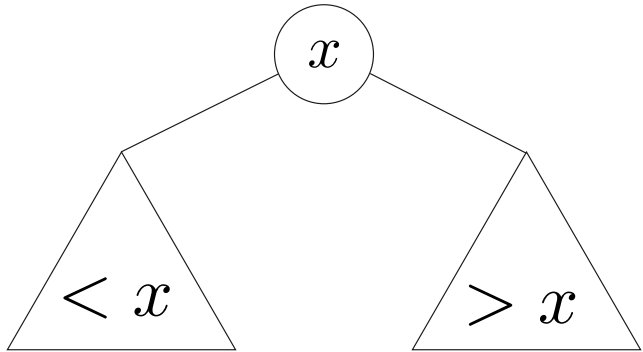
- Uma possível solução é utilizar uma árvore binária de pesquisa auto balanceável, como: Treaps, árvores-AVL, árvores-rubro-negras.
- Cada nó deve armazenar o tamanho da subárvore, a chave e o número de elementos igual a chave.
- Para cada elemento da entrada, inserimos o elemento da árvore e fazemos a busca por quantos elemento são maiores que ele.

Fila da padaria: solução

BST

Lembrando da estrutura de uma BST:

- A subárvore da esquerda possui elementos menores que x.
- a subárvore da direita possui elementos maiores que x.



Fila da padaria: solução

Seja y um número qualquer. Queremos saber quantos elementos maiores que eles na BST temos:

- Se $y < x$, então sabemos que a resposta corresponde ao tamanho da subárvore da direita mais ou número de ocorrências de x mais a solução recursiva sobre a subárvore da esquerda.
- Caso contrário, a resposta corresponde à solução do problema aplicado à subárvore da direita.

Fila da padaria: solução

Corpo principal

```
int main() {
    avl_tree_t *tree = nullptr;
    avl_tree_initialize(&tree);
    int n,p;
    ios::sync_with_stdio(false);
    cin >> n >> p;
    for(int i=0;i<n;i++){
        int x;
        cin >> x;
        x = x/p;
        avl_tree_insert(tree,x);
        cout << avl_tree_get_greater(tree,x) << (i < n-1 ? ' ' : '\n');
    }

    avl_tree_delete(&tree);
    return 0;
}
```

Fila da padaria: solução

Busca

```
int avl_tree_get_greater(avl_tree_t* t, int data){
    return avl_tree_get_greater_helper(t->root,data);
}
```

Fila da padaria: solução

Busca

```
int avl_tree_get_greater_helper(avl_node_t* v,int data){
    if(v==nullptr)
        return 0;
    if(data < v->data){
        return v->count + avl_tree_sz(v->right) +
            avl_tree_get_greater_helper(v->left,data);
    }
    return avl_tree_get_greater_helper(v->right,data);
}
```

Fila da padaria: análise

Análise de pior-caso

Cada busca possui tempo de pior caso $O(\lg n)$, em que n é o tamanho da entrada. Como são inseridos n elementos, o tempo de pior caso da solução é $\Theta(n \lg n)$.

Modelagem do problema

Dada a capacidade M da mochila e N itens com peso d_i e valor e_i, escolher zero ou mais itens com índices $i_1, i_2, \ldots, i_k$ de modo que

$$\sum_{j=1}^{k} d_{i_j} \leq M$$

e que a soma

$$S = \sum_{j=1}^{k} e_{i_j}$$

seja a maior possível.

Solução

- Conforme apresentado na modelagem, este problema corresponde ao da mochila binária.
- Seja $dp(i, t)$ a maior soma de fatores engajamento possível escolhendo zero ou mais elementos a partir do i-ésimo clipe, cuja soma das durações é menor ou igual a t segundos.
- O caso base é dado por $dp(N, t) = 0$, ou seja, sem nenhuma opção de escolha (sufixo vazio), a soma máxima é zero.
- Para $i > 0$ há duas transições possíveis. A primeira delas é

$$dp(i, t) = dp(i + 1, t),$$

que significa que o i-ésimo clipe não foi selecionado.

Solução

- Caso $t_i \leq t$, há uma segunda transição possível, dada por

$$dp(i, t) = dp(i + 1, t - t_i) + e_i,$$

  correspondente à escolha do i-ésimo clipe.
- Para cada clipe, deve ser escolhida, dentre as transições possíveis, a que maximiza o valor de $dp(i, t)$.
- A resposta do problema é dada por $dp(0, 60 \times M)$, pois M é dado em minutos e as durações d_i são dadas em segundos.

Gama, sempre Gama!: solução

```
long long dp(int i, int t, int N, const vector<ii>& xs)
{
    if (i == N)
        return 0;

    if (st[i][t] != -1)
        return st[i][t];

    auto res = dp(i + 1, t, N, xs);
    auto [d, e] = xs[i];

    if (t >= d)
        res = max(res, dp(i + 1, t - d, N, xs) + e);

    return (st[i][t] = res);
}
```

Gama, sempre Gama!

Análise de pior-caso

Como há $\Theta(NM)$ estados e as transições são feitas em $\Theta(1)$, esta solução tem complexidade $\Theta(NM)$.

Hortaliças

Definição (Distância entre reta e ponto)

A distância entre a reta $r = ax + by + c$ e o ponto $P = (x, y)$ é dada por

$$\text{dist}(r, P) = \frac{|ax + by + c|}{\sqrt{a^2 + b^2}}$$

Hortaliças

Modelagem do problema

Dados N pontos interiores ao paralelogramo $ABCD$, determine o segmento de reta r, paralelo aos segmentos a AB e CD e posicionado entre ambos, que minimiza a função

$$d(r, N) = \sum_{i=1}^{N} \mathrm{dist}(r, P_i)$$

Hortaliças

Solução

- Seja $M = D_y - A_y$. Uma solução que avalia todas as $M - 1$ retas possíveis tem complexidade $O(MN)$ e leva ao veredito TLE.
- Para reduzir a complexidade da solução, observe que, como todas as retas são paralelas, os coeficientes a e b serão idênticos em todas as possíveis retas.
- Assim, a função a ser minimizada pode ser simplificada para

$$d(r, N) = \sum_{i=1}^{N} |ax_i + by_i + c_r|,$$

dispensando o cálculo da raiz quadrada, o uso de frações e de aritmética de ponto flutuante.

Hortaliças

Solução

- Agora, considere um segmento de reta r fixo, com coeficiente c_r.
- Para $c_s = c_r + 1$, há dois cenários possíveis que relacionam $d(s, N)$ com $d(r, N)$:
  1. se $c_r \geq -(ax_i + by_i)$, então $|ax_i + by_i + c_s| = |ax_i + by_i + c_r| + 1$;
  2. se $c_r < -(ax_i + by_i)$, então $|ax_i + by_i + c_s| = |ax_i + by_i + c_r| - 1$;
- Seja $\delta_r(P_i)$ a função delta de Kronecker tal que $\delta_r(P_i) = 1$ se $c_r \geq -(ax_i + by_i)$, ou zero, caso contrário.
- Seja $\mu_r(P_i)$ uma função tal que $\mu_r(P_i) = 1$ se $c_r < -(ax_i + by_i)$, e zero, caso contrário.

Hortaliças

Solução

- Logo,

$$\delta(r, N) = \sum_{i=1}^{N} \delta_r(P_i) - \mu_r(P_i)$$

  computa a variação de $d(s, N)$ em relação a $d(r, N)$.
- Como $\delta(r, N)$ é monótona não-decrescente em relação ao coeficiente c_r, a função $d(r, N)$ é unimodal.
- Portanto, a reta que minimiza $d(r, N)$ pode ser determinada por meio de uma busca ternária nos coeficientes c das retas candidatas a solução.

Hortaliças

```
struct Line {
    ll a, b, c;

    Line(const Point& P, const Point& Q)
        : a(P.y - Q.y), b (Q.x - P.x), c(P.x * Q.y - Q.x * P.y)
    {
    }

    ll distance(const Point& p) const        // Distância de p à reta
    {
        return llabs(a*p.x + b*p.y + c);
    }
};
```

Hortaliças

```
ll distances(const Line& r, const vector<Point>& ps)
{
    ll sum = 0;

    for (auto& p : ps)
        sum += r.distance(p);

    return sum;
}
```

Hortaliças

```
auto solve(const Point& A, const Point& B,  const Point&, const Point& D, 
{
    ll a = 1, b = D.y - A.y - 1, ans = a;
    auto best = oo;

    while (a <= b)
    {
        auto m1 = a + (b - a)/3;
        auto m2 = b - (b - a)/3;

        auto r = Line(Point { A.x, A.y + m1 }, Point { B.x, B.y + m1 });
        auto s = Line(Point { A.x, A.y + m2 }, Point { B.x, B.y + m2 });

        auto dist1 = distances(r, ps);
        auto dist2 = distances(s, ps);
```

Hortaliças

```
            if (dist1 > dist2) {
                a = m1 + 1;

                if (dist2 < best) {
                    best = dist2; ans = m2;
                }
            } else {
                b = m2 - 1;

                if (dist1 < best) {
                    best = dist1; ans = m1;
                }
            }
        }

        return make_pair(Point { A.x, A.y + ans }, Point { B.x, B.y + ans });
    }
```

Hortaliças

Análise de pior-caso

Como são feitas $\Theta(\log M)$ buscas e em cada uma delas são avaliados N pontos, esta solução tem complexidade $\Theta(N \log M)$.

Indo ao Mercado

Modelagem do problema

Seja d_i a quantidade de itens a serem adquiridos no dia, sendo que no dia i são vendidos a_i itens ao custo unitário de c_i. Um item que não foi adquirido no dia i pode ser adquirido no dia j pelo custo

$$C_j = c_i + \sum_{k=i}^{j-1} l_k$$

O problema consiste em adquirir $D = \sum_i d_i$ itens tal que a soma dos custos de aquisição de cada item seja minimizada.

Indo ao Mercado

Solução

- A solução do problema consiste em, a cada dia i, adquirir d_i pelo menor custo possível
- Se os custos dos itens que estão na geladeira forem computados usando a fórmula da modelagem do problema, a solução terá veredito TLE
- Então é preciso computar estes custos de forma eficiente
- Isto pode ser feito usando modelando a geladeira como um *Venice Set*
- Esta estrutura suporta 3 operações: adicionar um item ao conjunto (`add()`), somar `v` em todos os elementos (`update_all()`) e extrair o menor elemento do conjunto (`extract_min()`)

Indo ao Mercado: Venice Set

```
using ii = pair<int, int>;

struct VeniceSet
{
    multiset<ii> ms;
    ll acc = 0;

    void add(ll x, ll qtd) { ms.insert(ii(x - acc, qtd)); }

    void update_all(ll v) { acc += v; }

    ii extract_min()
    {
        auto [v, qtd] = *ms.begin();
        ms.erase(ms.begin());

        return { v + acc, qtd };
    }
};
```

Indo ao Mercado

Solução

- Seja M o número de elementos no *Venice Set*
- As operações de adição e extração do mínimo tem complexidade $O(\log M)$
- A operação de atualização tem complexidade $O(1)$
- Usando esta estrutura para modelar a geladeira, basta avaliar os itens na ordem da entrada
  1. Coloque o i-ésimo item na geladeira
  2. Extraia da geladeira os d_i itens de menor custo e some estes custos à resposta
  3. Atualize o acumulador da geladeira com o valor l_i

Indo ao Mercado: solução

```
auto solve(int N, const vector<ll>& ds, const vector<ll>& as,
                  const vector<ll>& cs, const vector<ll>& ls)
{
    ll ans = 0;
    VeniceSet s;

    for (int i = 0; i < N; ++i)
    {
        auto a = as[i];
        auto c = cs[i];
        auto d = ds[i];
        auto l = ls[i];

        s.add(c, a);
```

Indo ao Mercado: solução

```
        while (d > 0) {
            auto [cost, qtd] = s.extract_min();
            auto k = min(d, qtd);

            ans += k * cost;
            d -= k;
            qtd -= k;

            if (qtd > 0)
                s.add(cost, qtd);
        }

        s.update_all(l);
    }

    return ans;
}
```

Indo ao Mercado

Análise de pior-caso

Como há $\Theta(N)$ item a serem processado, e cada item é processado em $\Theta(\log N)$, esta solução tem complexidade $\Theta(N \log N)$.

Definição (Troca (*swap*))

Dado uma sequência de elementos $a_1, a_2, \ldots, a_N$, uma troca (*swap*) é uma operação que troca os valores de dois elementos adjacentes. Em outras palavras, se $i = 1, 2, \ldots, N-1$, uma troca envolvendo os elementos a_i e a_{i+1} transforma a sequência a_k original na sequência

$$b_k = a_1, a_2, \ldots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \ldots, a_N$$

Kart Indoor

Modelagem do problema

Dada uma sequência de N elementos, determine o número mínimo de trocas necessárias para ordenar seus elementos.

Kart Indoor

Solução

- Observe que as numerações dos karts seguem a ordem de largada, isto é, $1, 2, \ldots, N$.
- A partir da ordem de chegada dos karts, basta determinar a quantidade de trocas que serão efetuadas para recuperar a ordem de largada (ordem crescente)
- Como $N \leq 5 \times 10^3$, é possível contar o número de trocas adicionando um contador na implementação do *bubble sort*, obtendo uma solução $O(N^2)$
- É possível resolver este problema com complexidade $O(N \log N)$ adicionando um contador no *mergesort*

Kart Indoor – *Bubble sort*

```
auto solve(int N, vector<int>& karts)
{
    auto swaps = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int j = n - 1; j > i; j--) {
            if (karts[j - 1] > karts[j]) {
                swap(karts[j], karts[j - 1]);
                ans++;
            }
        }
    }

    return ans;
}
```

Kart Indoor – *Mergesort*

```
ll inversions(int a, int b, vector<int>& xs) {
    if (b - a == 1)
        return 0;

    auto m = b + (a -  b)/2;
    auto sum = inversions(a, m, xs) + inversions(m, b, xs);
    auto x = a, y = m, idx = 0;
    vector<int> temp(b - a);

    while (x < m and y < b) {
        if (xs[x] <= xs[y])
            temp[idx++] = xs[x++];
        else {
            temp[idx++] = xs[y++];
            sum += (m - x);
        }
    }
```

Kart Indoor – *Mergesort*

```
    while (x < m)
        temp[idx++] = xs[x++];

    while (y < b)
        temp[idx++] = xs[y++];

    for (int i = a; i < b; ++i)
        xs[i] = temp[i - a];

    return sum;
}
```

Kart Indoor

Análise de pior-caso

A solução baseada no *bubble sort* tem complexidade $O(\Theta N^2)$; a solução baseada no *mergesort* tem complexidade $O(\Theta N \log N)$.

Luka, Miku e Chocolate

Modelagem do problema

Dado um grafo direcionado $G = (V, E)$, o problema quer responder Q consultas do tipo: existe caminho de u para v?

Observação

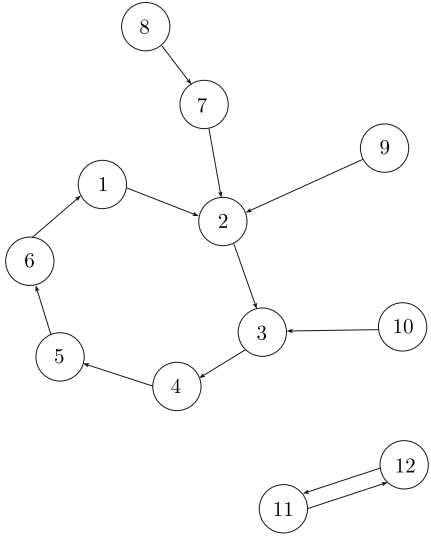
Uma restrição importante que o problema possui é que cada nó possui, **no máximo**, uma aresta saindo dele.
Esta restrição guiará o projeto de um algoritmo eficiente.

Luka, Miku e Chocolate: solução

- Primeiramente, iremos transformar o grafo direcionado G em um DAG (grafo acíclico direcionado) $G' = (V', E')$ de suas componentes fortemente conexas.
- Esse processo pode ser realizado em tempo $\Theta(|V| + |E|)$ usando o conhecido algoritmo de Tarjan.

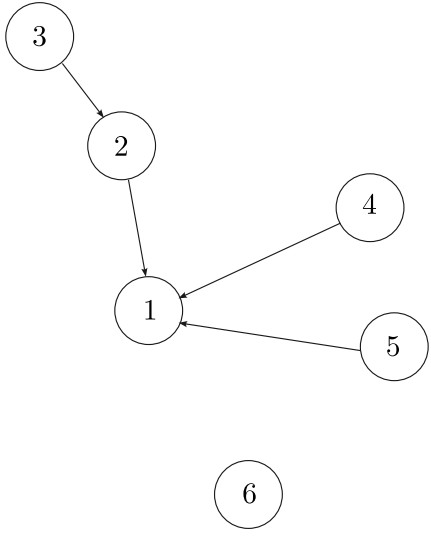
Luka, Miku e Chocolate: solução

Identificação da SCCs



Luka, Miku e Chocolate: solução

Identificação da SCCs

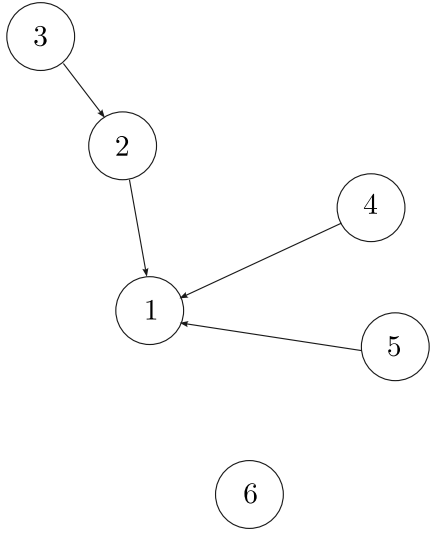


Luka, Miku e Chocolate: solução

- Agora a restrição do problema se faz fundamental para que consigamos resolvê-lo eficientemente.
- Temos vários caminhos que podem se encontrar, **mas nunca haverá uma bifurcação** após o encontro.
- Se invertermos as arestas de G', obteremos uma floresta G'^R

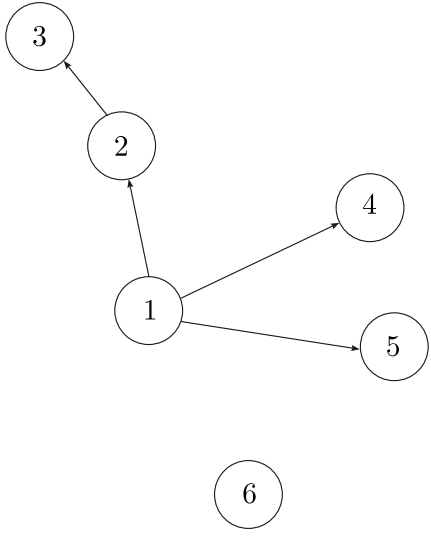
Luka, Miku e Chocolate: solução

Transformação do DAG em floresta



Luka, Miku e Chocolate: solução

Transformação do DAG em floresta

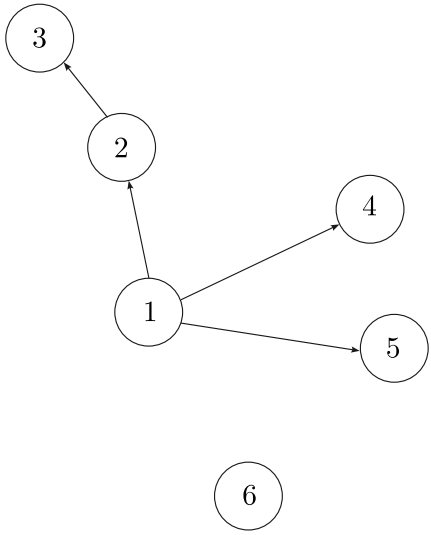


Luka, Miku e Chocolate: solução

- Para conectar todas as florestas, criaremos um novo nó em G'^R que possui uma aresta para a raiz de cada árvore.
- Isso facilitará na hora de responder cada consulta.
- Agora temos uma árvore na qual cada subárvore conectada a esse nó criado corresponde a uma árvore da floresta original.

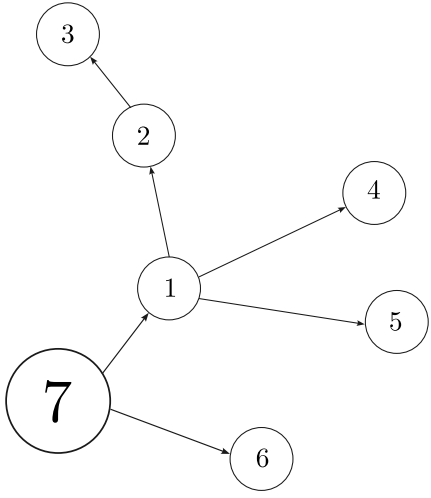
Luka, Miku e Chocolate: solução

Conexão das florestas



Luka, Miku e Chocolate: solução

Conexão das florestas



Luka, Miku e Chocolate: solução

- Agora, para saber se existe um caminho de u a v em G, basta respondermos se existe um caminho da componente em que v se encontra, digamos, v', até a componente em que u se encontra, digamos u', na árvore G'^R.
- Existe um caminho de v' até u' se v' é o ancestral comum mais baixo (LCA) de v'.
- A raiz que conecta as florestas tem o propósito de definir o LCA para qualquer par de nós.
- Utilizando árvores de segmentos, conseguimos responder a consulta de LCA em tempo logarítmico.

Luka, Miku e Chocolate: solução

Corpo principal

```
void solve() {
  queue<int> root_nodes;
  stack<int> scc_nodes;
  num.assign(n, -1);
  low.assign(n, n + 1);
  scc_num = 0;
  scc.assign(n, -1);
  visited.assign(n, WHITE);
```

Luka, Miku e Chocolate: solução

Corpo principal

```
  for (int u = 0; u < n; u++) {
    if (visited[u] == WHITE) {
      tarjan(u, scc_nodes);
    }
  }
  build_scc_tree();
  LCA lca(scc_tree, scc_num);
  for (const auto &[u, v] : queries) {
    if (lca.lca(scc[u], scc[v]) == scc[v]) {
      cout << "Sim\n";
    } else {
      cout << "Nao\n";
    }
  }
}
```

Luka, Miku e Chocolate: solução

Construção do DAG de SCCs

```
void tarjan(int u, stack<int> &scc_nodes) {
  visited[u] = GRAY;
  scc_nodes.push(u);
  num[u] = dfs_num;
  low[u] = dfs_num++;
  for (auto v : adj[u]) {
    if (visited[v] == WHITE) {
      tarjan(v, scc_nodes);
      low[u] = min(low[u], low[v]);
    } else if (visited[v] == GRAY) {
      low[u] = min(low[u], num[v]);
    }
  }
```

Luka, Miku e Chocolate: solução

Construção do DAG de SCCs

```
  if (low[u] == num[u]) {
    int w = -1;
    do {
      w = scc_nodes.top();
      scc_nodes.pop();
      scc[w] = scc_num;
      visited[w] = BLACK;
    } while (w != u);
    scc_num++;
  }
}
```

Luka, Miku e Chocolate: solução

Construção da árvore de SCCs

```
void build_scc_tree() {
  vector<vector<int>> scc_tree_inverted(scc_num);
  vector<int> in_degree(scc_num, 0);
  for (int u = 0; u < n; u++) {
    for (auto v : adj[u]) {
      if (scc[u] != scc[v]) {
        scc_tree_inverted[scc[u]].push_back(scc[v]);
      }
    }
  }
```

Luka, Miku e Chocolate: solução

Construção da árvore de SCCs

```
scc_tree.resize(scc_num + 1);
for (int u = 0; u < scc_num; u++) {
  for (auto v : scc_tree_inverted[u]) {
    scc_tree[v].push_back(u);
    in_degree[u]++;
  }
}
```

Luka, Miku e Chocolate: solução

Construção da árvore de SCCs

```
  // artificial node
  for (int i = 0; i < scc_num; i++) {
    if (in_degree[i] == 0)
      scc_tree[scc_num].push_back(i);
  }
}
```

Luka, Miku e Chocolate: análise

Análise de pior-caso

A complexidade da solução é $\Theta((|V| + |E|) + (Q \cdot t_{\mathrm{LCA}}))$, sendo Q o número de consultas.
Se as consultas de LCA forem implementadas através de árvores de segmentos, então $t_{\mathrm{LCA}} \in \Theta(\lg |V'|)$.

Luka, Miku e Chocolate: solução otimizada

Resolvendo em tempo ótimo

- É possível resolver o problema em tempo $\Theta(|V| + |E| + Q)$.
- Como saber se um nó é descendente de outro em uma árvore?
- Basta comparar os tempos de entrada e saída em uma DFS!
- Se um nó u começa a ser processado antes do nó v e termina de ser processado após o nó v é porque u é ancestral de v na árvore.
- Essa comparação leva tempo $\Theta(1)$.

Luka, Miku e Chocolate: solução otimizada

Resolvendo em tempo ótimo

```
void dfs(int u) {
  first[u] = dfs_num++;
  for (auto v : scc_tree[u]) {
    if (first[v] == 0) {
      dfs(v);
    }
  }
  last[u] = dfs_num++;
}
```

Luka, Miku e Chocolate: solução otimizada

Resolvendo em tempo ótimo

```
first.assign(scc_num+1, 0);
last.assign(scc_num+1, 0);
dfs_num = 1;
dfs(scc_num);
for (const auto &[u, v] : queries) {
  if (first[scc[v]] <= first[scc[u]] and last[scc[v]] >= last[scc[u]]) {
    cout << "Sim\n";
  } else {
    cout << "Nao\n";
  }
}
```

Autoria dos problemas

A) Alazão – Eric Grochowicz – UDESC
B) Balanceando Competições – Gustavo Machado Leal – UFG
C) Cidade Planejada – Vinicius Ruela Pereira Borges – UnB
D) Dobradinha da Ceilândia – Eduardo Freire dos Santos– UnB
E) Estrutural – Daniel Saad Nogueira Nunes – IFB
F) Fila da Padaria – Caleb Martim de Oliveira – UnB
G) Gama, sempre Gama! – Edson Alves da Costa Júnior – UnB/FGA
H) Hortaliças – Edson Alves da Costa Júnior – UnB/FGA
I) Indo ao mercado – Gustavo Machado Leal – UFG
J) Jogo da moeda – Eduardo Schwarz Moreira – UDESC
K) Kart Indoor – Vinicius Ruela Pereira Borges– UnB
L) Luka, Miku e chocolate – Caleb Martim de Oliveira – UnB

Revisores

- Alberto Tavares Duarte Neto – UnB
- Caleb Martim de Oliveira – UnB
- Daniel Saad Nogueira Nunes – IFB
- Edson Alves da Costa Júnior – UnB/FGA
- Gustavo Machado Leal – UFG

Sistemas

- Bruno César Ribas – UnB/FGA
- Daniel Saad Nogueira Nunes – IFB

Staff

- Ana Joyce Guedes
- Eduardo Ferreira Marques Cavalcante
- Maria Eduarda Carvalho

Organização CSBC

- Guilherme Novaes Ramos
- Lucy Mari Tabuti
- Vinicius Ruela Pereira Borges

Animeitor

- Emílio Wuerges.

Upsolving

https://moj.naquadah.com.br/cgi-bin/tag.sh/v_maratona_de_inverno_2024