2º Volume - Edição 2026

REVISTA MARATONA SBC DE PROGRAMAÇÃO

Uberlândia - 05 à 08 de Novembro 2026

31ª MARATONA SBC
(DE PROGRAMAÇÃO 2026)

EDIÇÃO DA FINAL BRASILEIRA 2026

Artigos técnicos, análises demográficas, maratonas pelo Brasil e muito mais



EXPEDIENTE

A Revista Maratona SBC de Programação é uma publicação anual da Diretoria de Competições Científicas da SBC publicada durante as finais brasileiras da Maratona SBC de Programação, voltada para os participantes da final e outros leitores interessados em programação competitiva.

PRESIDENTE DA SBC
THAIS VASCONCELOS BATISTA (UFRN)

VICE-PRESIDENTE
CRISTIANO MACIEL (UFMT)

Os artigos publicados nesta edição são de responsabilidade dos autores e não representam necessariamente a opinião da SBC.

CAPA:
John Curt Barros

COMITE EDITORIAL:
Bruno Monteiro
Lucas Morais
Mariana Gomes
Matheus Augusto
Rafael Grandsire
Thailsson Clementino

COLUNISTAS E COLABORADORES:
Amy Kuhn Hammes,
André L. P. Guedes,
Arthur Botelho,
Bernardo Amorim,
Bruno Ribas,
Daniel de P. Porto,
Daniel S. N. Nunes,
Edson A. da C. Júnior,
Fabrício Barbosa Viegas,
Gabriel P. Falcão,
Guilherme N. Ramos,
Ivan Henrique Costa Petrin
Kaio Vieira,
Leonardo Riether,
Lucas Pavanelli,
Miguel Miní,
Mariana G. Ramires,
Paulo Correia,
Pedro V. Sousa da Silva,
Racsó Galván,
Vinicius R. P. Borges,

DIREÇÃO DE IMAGEM E DESIGN:
John Curt Barros

ÍNDICE

A INTERPRETAÇÃO DE AUTÔMATOS PARA O ALGORITMO KMP 4
ARTHUR BOTELHO, UNB
LEONARDO RIETHER, UNB

APRENDIZAGEM NA ERA DO OFFLOAD COGNITIVO E O PAPEL DA PROGRAMAÇÃO COMPETITIVA 7
GABRIEL P. FALCÃO, UFMS
MARIANA G. RAMIRES, UFMS

DA SUBREGIONAL AO PÓDIO: PERFIS DE EDITORES EM DOIS CICLOS DA MARATONA SBC 12
BRUNO RIBAS, UNB

INTRODUÇÃO À OTIMIZAÇÃO POR DIVISÃO E CONQUISTA 14
PEDRO V. SOUSA DA SILVA, UFPR
ANDRÉ L. P. GUEDES, UFPR

DESAFIOS PARA A RECONSTRUÇÃO DE UM ESPÍRITO COMPETITIVO: A HISTÓRIA RECENTE DA MARATONA SBC NA UNIVERSIDADE FEDERAL DE PELOTAS 20
FABRÍCIO BARBOSA VIEGAS, UFPEL
AMY KUHN HAMMES, UFPEL

PROCESSAMENTO EM BATCHES COM DECOMPOSIÇÃO EM RAIZ QUADRADA 24
PAULO CORREIA, UFPE

PUSH-RELABEL 29
BERNARDO AMORIM, UFMG
KAIO VIEIRA, UFMG

HARDWARE DAS SEDES DA MARATONA SBC: POLÍTICA, REALIDADE BRASILEIRA E PANORAMA LATINO-AMERICANO 32
BRUNO RIBAS, UNB

RADIX HEAP: ACELERANDO O DIJKSTRA COM PESOS INTEIROS PEQUENOS 38
LUCAS PAVANELLI, FGV EMAP

TEORIA DE ROLES PARA FORMAR EQUIPES DE ICPC 40
RACSÓ GALVÁN, UNICAMP
MIGUEL MINÍ, UNICAMP

UM PROBLEMA INTERESSANTE SOBRE MEDIANAS 45
IVAN HENRIQUE COSTA PETRIN, UNICAMP

UNB NA FINAL NACIONAL DA MARATONA SBC: 2020-2025 49
BRUNO C. RIBAS, UNB
DANIEL DE P. PORTO, UNB
DANIEL S. N. NUNES, IFB
EDSON A. DA C. JÚNIOR, UNB
GUILHERME N. RAMOS, UNB
VINICIUS R. P. BORGES, UNB

A INTERPRETAÇÃO DE AUTÔMATOS PARA O ALGORITMO KMP

Arthur Botelho, UnB
Leonardo Riether, UnB

Resumo

As ideias aqui apresentadas são fruto de uma colaboração e já foram publicadas no Codeforces [1] e anteriormente no blog pessoal de Leonardo Riether [4]. O objetivo deste artigo é trazer o que acreditamos ser uma das formas mais naturais e intuitivas de se compreender o algoritmo Knuth-Morris-Pratt (KMP), utilizando o conceito de Autômatos Finitos, para a comunidade de programação competitiva brasileira.

1. Fundamentos e o Problema

Assumiremos que o leitor possui compreensão básica sobre a terminologia de grafos e cadeias (*strings*). Para entender o algoritmo, é necessário compreender conceitos fundamentais sobre Autômatos Finitos Não-Determinísticos (NFA) e Determinísticos (DFA).

De maneira simplificada, um autômato pode ser visto como um grafo direcionado onde:

1. Os vértices são chamados de **estados**. Há um estado inicial e um ou mais estados de *aceitação*.
2. As arestas representam **transições** e são rotuladas com caracteres de um alfabeto.

A simulação da leitura de uma cadeia S por um autômato pode ser visualizada através de *threads*, que trafegam de um estado a outro quando uma aresta compatível com o caractere atual de S é encontrada. Em um NFA, um estado pode ter múltiplas arestas saindo com o mesmo caractere (nesse caso, a *thread* se divide e segue vários caminhos simultaneamente), enquanto em um DFA, cada caractere leva a exatamente um estado.

Neste artigo, iremos mostrar o KMP como uma forma de resolver o problema clássico de Casamento de Cadeias: dadas duas cadeias S (texto) e P (padrão), encontre todas as subcadeias de S que correspondam exatamente a P.

2. Modelagem com Autômatos Não-Determinísticos

Existe uma construção simples de um NFA capaz de resolver esse problema. Para identificar ocorrências do padrão P, criamos um autômato de $|P|+1$ estados, organizados de forma sequencial e numerados de 0 a $|P|$. Cada estado i ($i < |P|$) possui uma aresta para o estado $i+1$ consumindo o caractere P_i. O estado 0 (inicial) possui uma transição (em laço) para si mesmo, consumindo qualquer caractere do alfabeto. O último estado ($|P|$) é o estado de aceitação.

Podemos visualizar, na Figura 1, como seria esse NFA para o padrão "abaa". O laço com o símbolo Σ no estado 0 indica o consumo de qualquer caractere do alfabeto.

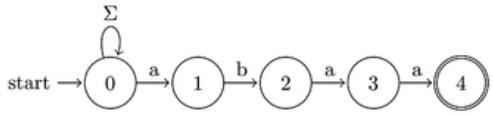


FIGURA 1: EXEMPLO DE UM NFA

Quando um texto é fornecido a este autômato, o laço no estado 0 gera uma nova thread a cada caractere lido, de forma que sempre pode-se iniciar um novo casamento. Ao mesmo tempo, threads geradas em passos anteriores avançam para o próximo estado se os caracteres corresponderem — caso contrário, a respectiva thread "morre". Se uma thread alcançar o estado de aceitação, um casamento foi encontrado. Note que isso já permite inclusive encontrar casamentos sobrepostos.

3. Exemplo Passo a Passo

Para facilitar a compreensão do comportamento simultâneo das threads, vamos simular a leitura do texto S = "abaa" pelo NFA do padrão P = "abaa". A evolução dos estados pode ser vista na Figura 2, onde os nós coloridos de vermelho representam as threads ativas.

Iniciamos com uma única thread no estado 0.

1. Lendo 'a': A thread no estado 0 se divide. Uma continua no estado 0 (devido ao laço Σ) e a outra avança para o estado 1. Temos threads em {0, 1}.

2. Lendo 'b': A thread no estado 1 avança para o estado 2. A thread no estado 0 permanece no 0. Temos threads em {0, 2}.

3. Lendo 'a': A thread no 2 avança para o 3. A thread no 0 se divide novamente, gerando uma no 1. Temos threads em {0, 1, 3}.

4. Lendo 'a': A thread no 3 avança para o 4, que é o estado de aceitação (portanto, encontramos um casamento). A thread que estava no 1 "morre" por não haver aresta com 'a', e o estado 0 gera uma nova no 1. Temos threads em {0, 1, 4}.

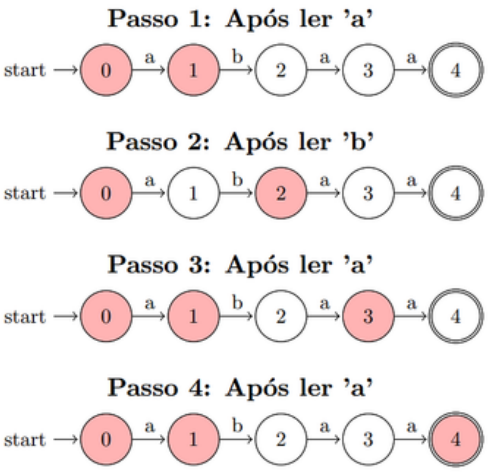


FIGURA 2: SIMULAÇÃO DO PADRÃO "abaa". OMITIMOS A ARESTA DE LAÇO NO ESTADO INICIAL PARA MAIOR CLAREZA.

Recomendamos o simulador de KMP de Leonardo Riether [5] para visualizar, de forma interativa, o processo de casamento com o autômato.

4. Custo da Simulação

A simulação ingênua deste NFA custaria tempo $\mathcal{O}(|S| \times |P|)$, já que múltiplas threads poderiam estar ativas e precisariam ser processadas a cada caractere do texto. No entanto, é possível otimizar esse processo observando a construção deste NFA.

5. Configurações Alcançáveis e a Thread Líder

Apesar da simulação não-determinística sugerir um número exponencial de configurações de threads simultaneamente ativas, pode-se provar que apenas $|P|+1$ configurações distintas são alcançáveis.

Denotaremos a thread mais avançada no autômato por "líder". Se o líder está no estado i, as posições de todas as outras threads (que estão em estados anteriores a i) são unicamente determinadas. Observando o Passo 3 do exemplo anterior, a thread líder está no estado 3. Note que as threads ativas à esquerda (nos estados 0 e 1) são uma consequência direta e previsível do caminho que o líder tomou.

Isso ocorre porque o caminho para alcançar o estado i exige que os últimos i caracteres lidos correspondam exatamente ao prefixo de tamanho i do padrão P. Logo, se mantivermos a posição da thread líder, sabemos que é possível inferir toda a informação necessária para o processamento do autômato, e que ocorrerá um casamento apenas quando o líder chegar ao estado $|P|$.

6. O Vizinho (Falha)

O desafio para simular o processamento do autômato surge quando a thread líder falha (o caractere lido não corresponde à próxima aresta e a thread morre). Nesse caso, precisamos saber quem assumiria a liderança.

A primeira candidata para novo líder será, naturalmente, a thread ativa mais avançada que não for o líder original. Chamaremos essa thread de "vizinho" — note que sempre há um vizinho desde que o líder esteja além do estado 0, pois este sempre possui uma thread ativa.

Assim que a thread líder falhar, verificamos se o seu vizinho consegue avançar com o caractere atual. Se sim, ele se torna o novo líder em sua respectiva próxima posição. Se também falhar, verificamos o vizinho do vizinho, e assim sucessivamente, até que ocorra um casamento ou a liderança volte ao estado 0.

Mesmo que precisemos retroceder a múltiplos vizinhos durante uma única falha, a soma total das quantidades de retrocessos durante o processamento de S no autômato que geramos baseado em P é $\mathcal{O}(|S|)$. O argumento é amortizado: no máximo uma thread é "criada" por caractere lido, logo, o número máximo de falhas (mortes de threads) está limitado ao tamanho de S.

7. Pré-cálculo e Implementação

Para simular o NFA em tempo $\mathcal{O}(|S|)$, precisamos pré-calcular um vetor vizinho, onde $vizinho[i]$ armazena o estado da thread vizinha caso o líder atual esteja no estado i.

O cálculo do vetor de vizinhos pode ser feito de forma construtiva iterando sobre o próprio padrão P. Trata-se essencialmente de fornecer P ao próprio NFA e verificar quem assumiria a liderança a cada passo.

Uma implementação do algoritmo descrito é apresentada no Código 1.

```cpp
struct KMP {
    string P;
    int n;  // n = |P|
    vector<int> vizinho;

    KMP(string& p) {
        P = p;
        n = (int)P.size();

        vizinho.resize(n + 1);
        vizinho[1] = 0;  // estado 0 sempre tem thread ativa

        for (int i = 1; i < n; i++)
            vizinho[i + 1] = proximo_lider(vizinho[i], P[i]);
    }

    bool avanca(int estado, char c) {
        return estado < n && P[estado] == c;
    }

    int proximo_lider(int lider, char entrada) {
        if (lider == 0)
            return P[0] == entrada;

        if (avanca(lider, entrada))
            return lider + 1;

        else
            return proximo_lider(vizinho[lider], entrada);
    }
};
```

CÓDIGO 1: IMPLEMENTAÇÃO DO ALGORITMO KMP ATRAVÉS DA ABORDAGEM DE AUTÔMATOS.

Esta estrutura pode ser usada para resolver o problema de Casamento de Cadeias em $\mathcal{O}(|S|+|P|)$. No Código 2, recebemos cadeias S e P e retornamos as posições (indexadas em 1) onde uma ocorrência de P em S termina.

```cpp
vector<int> ocorrencias(string P, string S) {
    KMP kmp(P);
    vector<int> res;
    int estado = 0;  // lider atual

    for (int i = 0; i < (int)S.size(); i++) {
        char c = S[i];
        estado = kmp.proximo_lider(estado, c);

        if (estado == (int)P.size())
            res.push_back(i + 1);
    }

    return res;
}
```

CÓDIGO 2: RESOLUÇÃO DO PROBLEMA DE CASAMENTO DE CADEIAS.

Para entender a complexidade de tempo deste algoritmo, podemos utilizar novamente análise amortizada. A cada caractere lido, a posição do líder é incrementada em no máximo uma unidade. Logo, ao processar uma cadeia de

tamanho N, ocorrem no máximo N incrementos. Por outro lado, cada chamada recursiva em proximo_lider (quando ocorre uma falha) substitui o líder atual pelo seu vizinho. Por definição, um vizinho de um estado é anterior a esse estado, então vizinho[lider] < lider. Assim, cada falha força a posição do líder a diminuir de valor. Como a posição não pode ser negativa (é limitada pelo estado inicial 0), o número total de diminuições em toda a execução não pode exceder o número total de incrementos. Portanto, o número total de transições ao longo do algoritmo é limitado por $2|S|$ na fase de busca e $2|P|$ no pré-cálculo, comprovando a complexidade de tempo $\mathcal{O}(|S| + |P|)$.

8. Relação com a Função de Prefixo

A interpretação mais comum para o algoritmo KMP em materiais de programação competitiva é através da Função de Prefixo (Prefix Function ou vetor π). Para cada prefixo p da cadeia, ela armazena o tamanho do maior prefixo próprio de p que também é sufixo de p. Isso é equivalente ao vetor $vizinho$ estudado nas últimas duas seções. Quando o líder está no estado i, a posição da segunda thread mais avançada é precisamente este maior prefixo-sufixo $\pi[i]$.

9. O Autômato Determinístico

O algoritmo mostrado até agora para o problema de Casamento de Cadeias tem complexidade $\mathcal{O}(|S|+|P|)$. No entanto, isso pode não ser o bastante para resolver alguns tipos de problemas.

A limitação da abordagem anterior surge do fato de que a complexidade de tempo depende de um argumento amortizado. Se estivermos processando um texto de forma linear, a quantidade de recuos nas *threads* é compensada pelos avanços. Porém, o argumento amortizado quebra quando precisamos ramificar o processamento. No contexto de programação competitiva, dois cenários comuns onde isso ocorre são:

1. Programação Dinâmica (PD) em cadeias: quando precisamos testar a adição de múltiplos caracteres diferentes ao final de uma cadeia, ramificando para próximos estados da PD.
2. Busca em Grafos onde uma cadeia é mantida: quando precisamos, para cada aresta do vértice atual da busca, ramificar para os vértices vizinhos com o caractere correspondente à aresta adicionado à cadeia mantida.

Nesses casos, calcular o próximo líder com o método amortizado pode resultar em uma complexidade ineficiente (causando Limite de Tempo Excedido). Para contornar isso, podemos pré-calcular, para todo líder e caractere do alfabeto, qual será o próximo líder, permitindo consultar as transições em $\mathcal{O}(1)$.

Isso nada mais é do que construir explicitamente o Autômato Determinístico (DFA) equivalente ao KMP. Em suma, iremos armazenar as transições diretas para todos os caracteres do alfabeto em qualquer estado. O Código 3 mostra uma extensão do Código 1 para gerar a matriz de transições, considerando caracteres de 'a' a 'z'.

```cpp
void constroi_dfa() {
    dfa = vector(n + 1, vector<int>(26));
    dfa[0][P[0] - 'a'] = 1;

    for (int k = 1; k <= n; k++) {
        for (int c = 0; c < 26; c++) {
            if (k < n && P[k] == 'a' + c)
                dfa[k][c] = k + 1;
            else
                dfa[k][c] = dfa[vizinho[k]][c];
        }
    }
}
```

CÓDIGO 3: CONSTRUÇÃO EXPLÍCITA DO AUTÔMATO DFA DO KMP.

Um excelente exemplo prático e recente da necessidade dessa estrutura ocorreu no problema B ("Busca Periódica") da Fase Zero da Maratona SBC de Programação de 2025 [3, 2]. O problema envolvia encontrar a periodicidade mínima de cadeias formadas por caminhos a partir da raiz em uma árvore [3].

Embora o problema em uma cadeia linear pudesse ser resolvido de forma eficiente apenas com o vetor $vizinho$, adaptar a abordagem clássica amortizada para uma DFS na árvore excede o tempo limite [2]. A solução oficial contorna essa ineficiência construindo exatamente o Autômato Determinístico do KMP [2]. Ao pré-calcular a matriz de transições para o alfabeto, cada passo da DFS avança o estado em $\mathcal{O}(1)$, garantindo que a solução execute no tempo esperado com complexidade $\mathcal{O}(26 \cdot N)$ [2].

10. Conclusão

Enxergar o algoritmo KMP não apenas como uma manipulação engenhosa de ponteiros de cadeias, mas como a simulação otimizada de um Autômato Finito, desmistifica profundamente o seu funcionamento e provas de corretude e complexidade. Além disso, essa visão também facilita o pensamento necessário para problemas avançados de PD e Grafos relacionados a cadeias onde o DFA pode ser aplicado.

Referências

[1] Arthur Botelho. The automaton interpretation for the kmp algorithm. Disponível em: https://codeforces.com/blog/entry/146191. Acesso em: 07 jul. 2026.
[2] Maratona SBC de Programação. Editorial da fase zero da maratona sbc de programação 2025. Disponível em: https://codeforces.com/gym/105925/attachments/download/31746/editorial-fase-zero-2025.pdf, 2025. Acesso em: 30 jul. 2026.
[3] Maratona SBC de Programação. Fase zero da maratona sbc de programação 2025 caderno de problemas. Disponível em: https://codeforces.com/gym/105925/attachments/download/31715/prova.br.pdf, 2025. Acesso em: 30 jul. 2026.
[4] Leonardo Riether. Kmp the automaton interpretation. Disponível em: https://leoriether.github.io/posts/kmp/. Acesso em: 07 jul. 2026.
[5] Leonardo Riether. Kmp simulator. Disponível em: https://leoriether.github.io/kmp-simulator/. Acesso em: 07 jul. 2026.

Aprendizagem na Era do Offload Cognitivo e o Papel da Programação Competitiva

Gabriel P. Falcão, UFMS
Mariana G. Ramires, UFMS

Resumo

Este artigo propõe uma reflexão crítica sobre a integração da Inteligência Artificial, em especial dos modelos de linguagem, ao processo de aprendizagem e à prática da computação. Discutimos os desafios impostos pelo offload cognitivo e apontamos a prática da programação competitiva como uma prática pedagógica de contraposição a essa nova era de acesso à informação.

1. Introdução

Nos últimos anos, a Inteligência Artificial (IA) passou a incorporar um papel de destaque em diferentes áreas, como a educação básica [1], o judiciário [2] e a medicina [3]. Diante desse cenário, não é surpreendente que na computação sua presença seja ainda mais disruptiva e difundida. Práticas de desenvolvimento assistido por IA deixaram de ser puramente experimentais e estão transformando a esteira de produção de software, da geração de código com modelos de linguagem de grande escala (LLMs) [4] até a automação da geração de testes [5].

Porém, à medida que a produtividade técnica atinge novos patamares, a comunidade científica começa a alertar para as consequências psicológicas e educacionais dessa automação. O termo offload cognitivo expressa o fenômeno em que o esforço analítico e o processo de raciocínio são integralmente terceirizados para a IA, reduzindo os estímulos cognitivos essenciais para o desenvolvimento cerebral [6].

Voltamos nossa atenção, então, para um passo antes da indústria: o ambiente de aprendizagem. À primeira vista, o ato de aprender e a dependência de ferramentas de IA, como os chatbots, parecem opostos, sobretudo sob a ótica do estímulo cognitivo. Torna-se essencial a valorização de ambientes que estimulem o raciocínio autônomo e de alta complexidade. É nesse cenário que a programação competitiva — com destaque para a Maratona SBC de Programação, que anualmente reúne milhares de estudantes de centenas de instituições de ensino brasileiras —, ganha uma nova dimensão pedagógica.

O objetivo deste texto é propor uma reflexão sobre como a programação competitiva atua como um ambiente essencial para o desenvolvimento do raciocínio computacional independente, contrapondo-se à tendência de terceirização cognitiva imposta pelas ferramentas de IA.

2. IA na Computação

Nos últimos anos, a Inteligência Artificial deixou de ser uma tecnologia emergente para tornar-se parte integrante do cotidiano da Computação. De acordo com o Stack Overflow Developer Survey 2025 [7], 84% dos respondentes utilizam ou planejam utilizar ferramentas de IA em seu fluxo de desenvolvimento, evidenciando a rápida consolidação dessas tecnologias na indústria.

Ferramentas baseadas em LLMs vêm sendo aplicadas em diversas atividades da Engenharia de Software, incluindo geração e compreensão de código, documentação, testes, depuração, manutenção e apoio ao processo de desenvolvimento como um todo [8]. Esse cenário torna inevitável discutir não apenas o impacto da IA no ciclo de vida do software, mas também seus efeitos sobre a formação dos futuros profissionais da Computação.

Entre as mudanças observadas na prática profissional, destaca-se o chamado vibe coding, abordagem em que desenvolvedores utilizam modelos de IA como participantes ativos do processo de programação, concentrando seus esforços na formulação dos problemas e na avaliação das soluções produzidas. Sob essa perspectiva, a Inteligência Artificial representa mais uma etapa no processo de abstração que acompanha a evolução da Computação. Assim como linguagens de alto nível, bibliotecas e frameworks reduziram a necessidade de lidar com detalhes de implementação, os sistemas atuais ampliam a produtividade ao automatizar atividades relacionadas ao próprio desenvolvimento de software.

Diante desse cenário, torna-se difícil imaginar a formação e a atuação profissional em Computação sem a presença dessas ferramentas. A questão, portanto, deixa de ser se a Inteligência Artificial deve ser utilizada e passa a ser como incorporá-la aos processos de aprendizagem sem comprometer o desenvolvimento das competências fundamentais da área.

3. Aprendizagem na era do offload cognitivo

A facilidade de acesso a respostas e soluções representa uma das principais vantagens das ferramentas modernas de Inteligência Artificial. Entretanto, essa mesma característica nos leva a refletir sobre a relação entre acesso à informação e construção do conhecimento.

A facilidade de acesso à informação não é um fenômeno novo, mas a Inteligência Artificial modifica sua escala e profundidade. Essa mudança intensifica um comportamento já observado por Sparrow, Liu e Wegner, segundo o qual indivíduos tendem a memorizar menos informações quando acreditam que poderão recuperá-las posteriormente por meios digitais, fenômeno conhecido como Google Effect [9]. Assim, a IA não apenas amplia o acesso à informação, mas também favorece a terceirização de etapas cognitivas tradicionalmente envolvidas na aprendizagem.

Um dos muitos usos da IA na computação é como ferramenta de aprendizado. Pereira Filho, Souza e Paula [10] realizaram experimentos qualitativos e quantitativos nos quais solicitaram ao ChatGPT que resolvesse questões comumente aplicadas em cursos de iniciação em programação. Embora o modelo tenha produzido respostas corretas em mais de 80% das instâncias, também foram identificados problemas como soluções excessivamente complexas ou desprovidas de explicação, o que pode ser prejudicial ao processo de aprendizado dos estudantes.

Além disso, Valovy e Buchalcevova [11] conduziram um estudo experimental com 38 estudantes de engenharia de software e 5 profissionais para avaliar o impacto do uso de IAs generativas (como ChatGPT e Copilot) no desenvolvimento. Nas entrevistas qualitativas subsequentes, conduzidas até a saturação dos dados, embora a percepção geral sobre as ferramentas tenha sido positiva, sobressaiu-se uma ressalva entre os discentes: a maioria dos alunos entrevistados (80%, ou 4 de 5) reportou que o aprendizado do conteúdo e dos fundamentos de programação seria mais efetivo sem a interferência inicial da IA.

Esses resultados não devem ser interpretados como uma rejeição ao uso da Inteligência Artificial no ensino. Pelo contrário, eles evidenciam uma questão mais profunda: a diferença entre ter acesso à informação e efetivamente transformá-la em conhecimento.

Sob a perspectiva da Ciência da Informação, informação e conhecimento não são conceitos equivalentes. Conforme argumenta Malheiro da Silva [12], a informação somente adquire significado quando é interpretada e apropriada pelo sujeito em um determinado contexto. Em outras palavras, o simples contato com uma resposta correta não garante sua compreensão, tampouco a capacidade de aplicá-la em novas situações.

Essa discussão torna-se ainda mais relevante em um cenário caracterizado pela abundância informacional. Satur e Malheiro da Silva [13] defendem que a sociedade contemporânea exige o desenvolvimento de competências que permitam localizar, compreender, avaliar criticamente e utilizar informações de forma efetiva. Assim, o desafio educacional deixa de ser apenas o acesso ao conteúdo e passa a envolver a capacidade de transformá-lo em conhecimento aplicável.

Ao aplicar essa reflexão na formação em Computação, observamos que grande parte da aprendizagem ocorre durante o próprio processo de resolução de problemas. A formulação de hipóteses, a identificação de erros, a comparação entre diferentes abordagens e o refinamento de soluções são etapas que contribuem para a construção do aprendizado e do pensamento computacional. Embora ferramentas de IA possam acelerar esse processo e oferecer suporte valioso aos estudantes, elas não eliminam a necessidade de participação ativa na construção do conhecimento.

Nesse contexto, torna-se necessário refletir sobre quais competências permanecem essenciais independentemente das tecnologias utilizadas.

4. A importância do raciocínio computacional

Se o acesso à informação e à geração de código torna-se cada vez mais simples, as competências exigidas dos profissionais da Computação também passam por uma transformação. Mais do que produzir código, espera-se que desenvolvedores sejam capazes de compreender problemas, avaliar diferentes estratégias de solução, interpretar requisitos e analisar criticamente os resultados produzidos por ferramentas inteligentes.

Nesse contexto, ganha destaque o conceito de raciocínio computacional, entendido como a capacidade de formular problemas de maneira que suas soluções possam ser descritas e executadas de forma sistemática. Esse processo envolve habilidades como abstração, decomposição de problemas, reconhecimento de padrões e construção de algoritmos — competências que permanecem essenciais independentemente da linguagem de programação ou das ferramentas utilizadas.

Paradoxalmente, acreditamos que o avanço da Inteligência Artificial torna essas habilidades ainda mais relevantes. Embora modelos de linguagem sejam capazes de gerar implementações cada vez mais sofisticadas, sua utilização efetiva depende da capacidade do usuário de definir corretamente o problema, avaliar a adequação das soluções propostas, identificar limitações e adaptar respostas a diferentes contextos. Em outras palavras, quanto menor o esforço necessário para produzir código, maior passa a ser a importância da compreensão dos problemas que esse código busca resolver.

Dessa forma, a formação em Computação não deve concentrar seus esforços apenas no domínio de novas ferramentas, mas também na criação de oportunidades para que estudantes desenvolvam o raciocínio computacional por meio da resolução ativa de problemas. É nesse contexto que ambientes voltados à prática sistemática da resolução de problemas adquirem papel fundamental na formação de futuros profissionais.

5. O Papel da Programação Competitiva

Diante da necessidade de desenvolver competências relacionadas ao raciocínio computacional, a Programação Competitiva apresenta-se como um ambiente de aprendizagem particularmente adequado para a formação em Computação. Diversos trabalhos têm investigado a Programação Competitiva como ferramenta de apoio ao ensino da área. No contexto brasileiro, Nogueira e Pozzer [14] analisaram um projeto preparatório para a Maratona SBC de Programação e observaram que os participantes relataram evolução tanto no desempenho competitivo quanto na percepção do próprio aprendizado.

Ao contrário de atividades baseadas na reprodução de exemplos ou na simples consulta a soluções prontas, a Programação Competitiva coloca o estudante diante de problemas inéditos, exigindo a interpretação cuidadosa do enunciado, a identificação das restrições, a formulação de

hipóteses e a construção gradual de uma solução. Durante esse processo, erros tornam-se parte natural da aprendizagem, levando o competidor a revisar estratégias, comparar diferentes abordagens e refinar seu raciocínio. Sob a perspectiva discutida na Seção 3, esse processo favorece a apropriação da informação e sua transformação em conhecimento efetivo.

As habilidades desenvolvidas nesse contexto vão além da implementação de algoritmos. Resolver problemas competitivos exige abstração, decomposição de problemas, reconhecimento de padrões, análise de complexidade e tomada de decisão sob restrições — competências diretamente relacionadas ao raciocínio computacional e que permanecem relevantes independentemente das ferramentas utilizadas para escrever código [15].

O crescimento da Inteligência Artificial não reduz a importância desse processo; ao contrário, reforça sua necessidade. Considere, por exemplo, um problema clássico envolvendo caminhos mínimos em grafos. Um modelo de linguagem pode rapidamente sugerir uma implementação correta do algoritmo de Dijkstra. Entretanto, identificar que o problema pode ser modelado como um grafo, compreender que as restrições permitem a utilização desse algoritmo, avaliar sua complexidade e verificar a correção da solução continuam sendo responsabilidades do estudante. A geração automática de código não substitui a compreensão do problema.

Isso não significa que a Inteligência Artificial deva ser excluída do processo de aprendizagem da Programação Competitiva. Durante o treinamento, essas ferramentas podem atuar como tutores, auxiliando na compreensão de conceitos, explicando algoritmos, sugerindo casos de teste e analisando implementações. Entretanto, a resolução efetiva dos problemas continua dependendo da participação ativa do estudante. Nas competições oficiais, inclusive, consultas externas são proibidas, fazendo com que o competidor dependa exclusivamente do próprio raciocínio para enfrentar os desafios propostos.

6. Considerações finais

Assim, compreende-se a programação competitiva como uma poderosa aliada do processo de ensino da computação em uma era propensa ao offload cognitivo. Em um cenário caracterizado pelo acesso imediato à informação e pela crescente automação da implementação de software, não se diminui a relevância do fator humano; pelo contrário, desloca-se o eixo de exigência profissional para a capacidade profunda de abstração e formulação de problemas. Nesse contexto, práticas como a Maratona SBC de Programação assumem o papel fundamental de oferecer um ambiente controlado que estimule os estudantes a desenvolverem as competências necessárias para compreender problemas, construir soluções e, consequentemente, utilizar ferramentas de IA de forma crítica, consciente e efetiva.

Referências

[1] Thiago da Silva Fialho, Christian de Vargas Silva, Fabrício Herpich, Marina Sérgio Carradore, Eliane Pozzebon. Inteligência artificial e educação básica: uma revisão sistemática das aplicações. TEyET, v. 41, 2025.
[2] Carolina Watanabe, Taynara França. O Impacto da Inteligência Artificial no Judiciário. Revista Em Tempo, v. 24, n. 1, p. 47–73, 2025.
[3] Vanessa Schmidt Bortolini, Alexandre de Souza Garcia, Wilson Engelmann. USO ÉTICO DA INTELIGÊNCIA ARTIFICIAL NA SAÚDE: UMA REVISÃO SISTEMÁTICA DA LITERATURA. Revista Jurídica Gralha Azul - TJPR, v. 1, n. 28, 2025.
[4] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, Sunghun Kim. A Survey on Large Language Models for Code Generation. ACM Trans. Softw. Eng. Methodol., v. 35, n. 2, 2026.
[5] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, Qing Wang. Software Testing With Large Language Models: Survey, Landscape, and Vision. IEEE Transactions on Software Engineering, v. 50, n. 4, p. 911–936, 2024.
[6] Michael Gerlich. AI Tools in Society: Impacts on Cognitive Offloading and the Future of Critical Thinking. Societies, v. 15, n. 1, 2025.
[7] Stack Overflow. 2025 Stack Overflow Developer Survey: AI, 2025. Disponível em: https://survey.stackoverflow.co/2025/ai. Acesso em: 7 jul. 2026.
[8] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. ACM Transactions on Software Engineering and Methodology, v. 33, n. 8, p. 220:1–220:79, 2024.
[9] Betsy Sparrow, Jenny Liu, Daniel M. Wegner. Google Effects on Memory: Cognitive Consequences of Having Information at Our Fingertips. Science, v. 333, n. 6043, p. 776–778, 2011.
[10] Luiz Pereira Filho, Talita Souza, Luciano Paula. Análise das Respostas do ChatGPT em Relação ao Conteúdo de Programação para Iniciantes. In: Anais do XXXIV Simpósio Brasileiro de Informática na Educação. SBC, p. 1738–1748, 2023.
[11] Marcel Valovy, Alena Buchalcevova. The Psychological Effects of AI-Assisted Programming on Students and Professionals, 2023.
[12] Armando Malheiro da Silva. A Informação: da Compreensão do Fenómeno e Construção do Objecto Científico. Porto: Edições Afrontamento, 2006.
[13] Roberto Vilmar Satur, Armando Malheiro da Silva. A Aprendizagem Visando a Competência em Informação na Sociedade em Tempos de Infoesfera. Perspectivas em Gestão & Conhecimento, v. 10, n. Especial, p. 2–22, 2020.
[14] Jonathan Weber Nogueira, Fernando Pedrazzi Pozzer. Uma análise dos resultados obtidos no primeiro ano do projeto preparatório para a Maratona de Programação. Revista ComInG, v. 7, n. 1, p. 31–38, 2023.
[15] Jeannette M. Wing. Computational Thinking. Communications of the ACM, v. 49, n. 3, p. 33–35, 2006.



PREMIUM

btg pactual

O BTG Pactual é o maior Banco de Investimentos da América Latina, e a busca constante pela excelência faz parte do nosso DNA. Para nós, trabalho duro e excelência acadêmica são parte do que vai moldar o futuro do país. Apoiar a Maratona SBC de Programação é investir nos estudantes que hoje mergulham de cabeça em desafios de alta complexidade e, amanhã, resolverão os problemas que importam no mercado e na tecnologia.

"Todo maratonista lembra da sensação da última hora de um contest: placar congelado, um problema que ninguém do time resolveu ainda e a decisão de continuar tentando em vez de aceitar o resultado que já está na mesa. Passei por isso durante cinco anos, com algumas medalhas, uma participação latino-americana, duas participações na etapa internacional da ICPC e incontáveis horas de treino com meu time. Na época, eu achava que estava aprendendo algoritmos. Descobri depois que o que ficou foi outra coisa: a disposição de não desistir de um problema difícil.

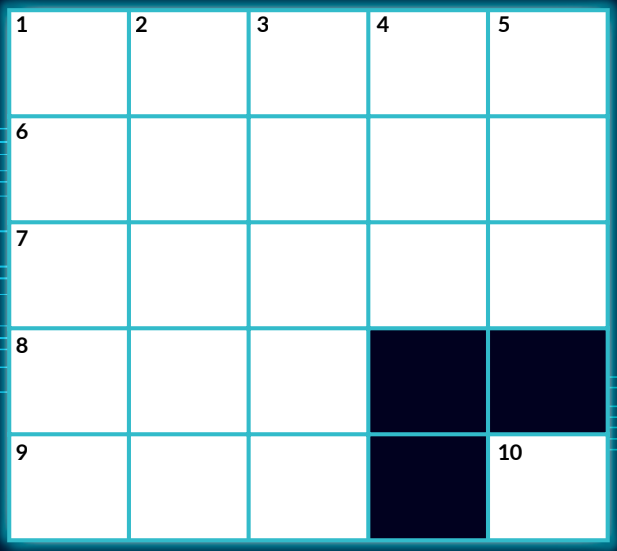
Aqui no BTG Pactual, excelência, trabalho em equipe e trabalho duro não são frases de parede, são o que separa uma entrega boa de uma entrega que resolve e impressiona. Excelência é o que se constrói depois de horas dedicadas a um problema de alta complexidade, quando a solução quase certa não basta e é preciso recomeçar do zero. Trabalho em equipe é reconhecer rápido quem tem a expertise que o problema pede, algo que se aprende quando são três pessoas, um computador e cinco horas em contagem. E trabalho duro é a convicção, construída na prática, de que mergulhar de cabeça no problema leva a resultados que antes não pareciam possíveis. Essa dedicação e essa persistência são o que a Maratona tem de mais raro a oferecer, e é o que me leva todos os dias ao próximo nível."

Vinícius Lettiéri Proença, associate director do BTG Pactual e ex-maratonista

PASSATEMPOS COMPETITIVOS

por Eric Grochowicz

Mini Cruzadinha Programação Competitiva 1



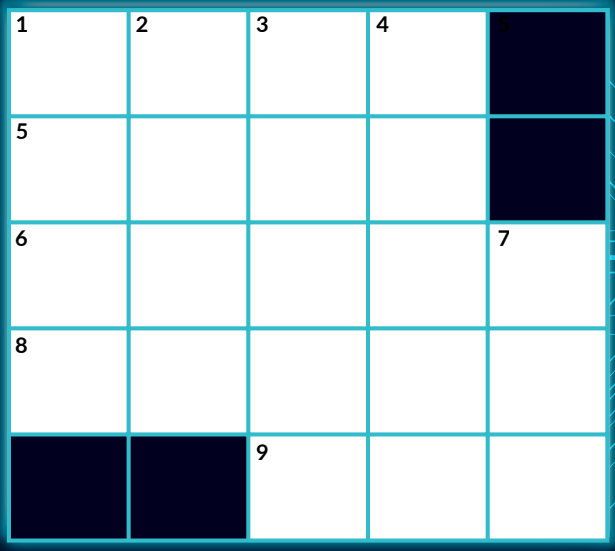
HORIZONTAL

1. Demonstrar orgulho próprio, celebrar-se.
6. Cheio, no Chile.
7. Saí de fininho.
8. Outro nome para a árvore de Fenwick.
9. Possível nome para a variável que guarda a resposta.
10. A letra mais frequente nesta mini cruzadinha.

VERTICAL

1. Solo; terreno que não foi loteado.
2. Grito frequente de Dave Seville no filme Alvin e os Esquilos.
3. Árvore de segmentos com consultas de soma e atualizações de máximo em intervalo.
4. Certa operação booleana.
5. Return On Investment.

Mini Cruzadinha Programação Competitiva 2



HORIZONTAL

1. Tipo de variável em C/C++ tipicamente maior que int.
5. Colocar junto.
6. Residir.
8. Um de três de certo algoritmo de strings.
9. Faça som.

VERTICAL

1. Lâmpada, em inglês.
2. Discordar de.
3. Esposas dos filhos.
4. Agradável, aprazível.
7. Runtime Error.



Solução Programação Competitiva 1
Horizontal: (1) GABAR; (6) LLENO; (7) EVADI; (8) BIT; (9) ANS; (10) A.
Vertical: (1), GLEBA; (2), ALVIN; (3), BEATS; (4), AND; (5), ROI.

Solução Programação Competitiva 2 Horizontal: (1) LONG; (5) APOR; (6) MORAR; (8) PRATT; (9) SOE. **Vertical:** (1), LAMP; (2), OPOR; (3), NORAS; (4), GRATO; (7), RE.

DA SUBREGIONAL AO PÓDIO: PERFIS DE EDITORES EM DOIS CICLOS DA MARATONA SBC

Bruno Ribas, UnB

Resumo

Este é um artigo informativo e descritivo: a partir da telemetria de uso de editores em seis competições da Maratona SBC (dois ciclos, três fases, mais de 3.000 times), ele sumariza um achado central. O Visual Studio Code domina em todas as fases (73% a 83% das equipes), mas entre os 30 melhores os IDEs robustos praticamente somem e o Vim sobe para 23% a 37%, padrão que chamamos de "vale dos IDEs". A análise por país ainda revela perfis culturais nítidos, com destaque para o Brasil como o país do Vim. Não buscamos relação causal entre ferramenta e desempenho.

1. Dados e método

A Direção de Sistemas mantém em cada máquina das sedes uma coleta automática de telemetria simples: tempo agregado em que cada editor ficou aberto e um instantâneo de CPU e RAM. Essa coleta roda sobre a infraestrutura padronizada das sedes: o sistema de julgamento BOCA [1] e o ambiente Maratona Linux [2, 3], que já traz pré-instalados todos os editores aqui analisados. Os placares oficiais são públicos (https://scorelatam.naquadah.com.br/).

Analisamos dois ciclos completos, 2024–25 e 2025–26, cada um com as três fases da competição: a Primeira Fase (a regional brasileira), a Final Latam e o PDA. As três fases de um mesmo ciclo não são amostras independentes, e sim um funil: os times que disputam a Final já passaram por uma Primeira Fase, e os que chegam ao PDA vieram da Final. A unidade que se repete é o ciclo: um achado vale mais quando reaparece nos dois, não por ser contado seis vezes. Em números, são 855, 495 e 40 times no ciclo 24–25 e 1.047, 578 e 52 no ciclo 25–26 (Primeira Fase, Final e PDA, nessa ordem), somando mais de 3.000 registros, um por time em cada evento.

A cobertura varia por fase. Na Primeira Fase, ainda que alguns países usem nossa regional, nem todos adotam o sistema centralizado, de modo que os dados de editores são essencialmente dos times brasileiros. Na Final Latam, distribuída em uma sede por país latino-americano (a brasileira é a Final Brasileira), todas rodam o Maratona Linux, garantindo cobertura completa. O PDA, em sede única, também oferece cobertura total: 40 times oficiais em 2025 e 41 em 2026, mais 11 equipes convidadas (CCL) em 2026. Assim, na Tabela 1 a coluna Reg é essencialmente brasileira, enquanto Fin e PDA abrangem toda a América Latina; é também o que torna confiável a análise por país.

Adotamos o critério dos relatórios internos: um editor é considerado **utilizado** por um time se permaneceu aberto por ao menos 60 minutos durante a prova, que dura 300 minutos. Um mesmo time pode usar mais de um editor.

2. Domínio de VS Code e crescimento do Vim

TABELA 1: ADOÇÃO DE EDITORES (% DE TIMES COM ≥ 60 MIN ABERTO) EM SEIS COMPETIÇÕES.

	Ciclo 24–25			Ciclo 25–26		
Editor	Reg	Fin	PDA	Reg	Fin	PDA
N (times)	855	495	40	1047	578	52
Visual Studio Code	74,9	73,5	82,5	82,0	79,6	80,8
gedit	23,7	27,5	37,5	24,3	23,4	5,8
Vim	5,4	6,5	27,5	5,3	6,1	19,2
PyCharm	13,7	7,7	2,5	12,4	10,9	3,8
CLion	8,8	12,1	5,0	12,5	16,4	1,9
IntelliJ IDEA	8,2	3,8	0,0	6,0	5,2	0,0
Code::Blocks	16,5	20,6	7,5	11,6	15,1	0,0

A Tabela 1 apresenta o panorama. O Visual Studio Code [4] é o editor mais usado em todas as competições, com adoção entre 73% e 83%, praticamente estável entre fases e ciclos. O gedit [5] oscila e é sensível à imagem do PDA: cai de 37,5% em 2025 para 5,8% em 2026, indicando que sua presença como *default* pesa na adoção; é prudente lê-lo como medida de **disponibilidade**, não de **preferência**.

O Vim [6] mostra um padrão mais informativo. Na Regional e na Final Latam sua adoção fica entre 5% e 7%, mas no PDA salta para 27,5% em 2025 e 19,2% em 2026, aumento de 3 a 5 vezes em ambos os ciclos. Como o Vim raramente é aberto por acidente, a leitura mais plausível é que o salto reflete a maior densidade de competidores experientes em sede única.

3. O "vale dos IDEs" entre os Top-30

TABELA 2: ADOÇÃO DE EDITORES (%) ENTRE AS 30 MELHORES EQUIPES DO PLACAR OFICIAL DE CADA EVENTO.

	Ciclo 24–25			Ciclo 25–26		
Editor	Reg	Fin	PDA	Reg	Fin	PDA
Visual Studio Code	56,7	80,0	83,3	73,3	76,7	76,7
gedit	16,7	23,3	43,3	30,0	30,0	6,7
Vim	23,3	36,7	33,3	26,7	23,3	23,3
PyCharm	0,0	0,0	0,0	0,0	0,0	0,0
CLion	3,3	0,0	3,3	10,0	3,3	3,3
IntelliJ IDEA	0,0	0,0	0,0	0,0	0,0	0,0
Code::Blocks	0,0	3,3	10,0	3,3	3,3	0,0

A Tabela 2 restringe a análise às 30 melhores equipes. Dois pontos se sobressaem. Primeiro, **IntelliJ IDEA e PyCharm [7] não aparecem no Top-30 de nenhum dos seis eventos**; CLion e Code::Blocks aparecem, mas com adoção máxima de 10% e tipicamente abaixo de 4%. A fração de equipes que usaram qualquer IntelliJ ou Code::Blocks, no conjunto completo, fica entre 35% e 37% nas Regionais e Finais, e despenca para 5,8% a 12,5% no PDA. Esse é o "vale dos IDEs", padrão que reaparece nas três fases dos dois ciclos.

Segundo, o Vim ocupa o espaço deixado pelos IDEs no topo: em todas as seis competições, sua adoção no Top-30 fica entre 23% e 37%, faixa estável que não se vê quando olhamos todos os times. Editores leves (gedit, Vim, Geany) são, portanto, sistematicamente sobre-representados entre os melhores; IDEs pesados são sub-representados. Não afirmamos causa: pode ser que competidores que treinam mais prefiram editores ágeis, e não que o editor os faça melhores.

4. Perfis culturais por país

TABELA 3: ADOÇÃO DE EDITORES (%) POR PAÍS, NA FINAL LATAM DE 2025 (FOCO EM PAÍSES COM ≥ 20 EQUIPES). VSC = VISUAL STUDIO CODE; PYC = PYCHARM; CLI = CLION; C::B = CODE::BLOCKS; GN = GEANY.

País (n)	VSC	gedit	Vim	CLi	PyC	C::B	Gn
México (133)	82,0	25,6	4,5	24,1	10,5	9,0	3,0
Bolívia (76)	77,6	31,6	0,0	14,5	7,9	42,1	6,6
Brasil (66)	81,8	27,3	21,2	6,1	1,5	0,0	6,1
Argentina (57)	75,4	10,5	5,3	3,5	1,8	8,8	22,8
Colômbia (41)	61,0	51,2	2,4	29,3	31,7	31,7	0,0
Chile (37)	83,8	13,5	8,1	8,1	5,4	0,0	0,0
Peru (22)	77,3	13,6	13,6	18,2	9,1	4,5	0,0
Cuba (22)	77,3	18,2	0,0	31,8	22,7	54,5	0,0

Cruzar editor com país revela que a aparente uniformidade do VS Code esconde perfis culturais bem distintos (Tabela 3). O **Brasil** apresenta um perfil singular: a maior adoção de Vim da Final (21,2%), combinada com nenhum uso de Code::Blocks (0%) e o menor uso de PyCharm (1,5%), exatamente o inverso do perfil de Cuba e Bolívia. Estas duas se inclinam ao **Code::Blocks** (54,5% e 42,1%). **Colômbia** é a mais "IntelliJ" (29,3% CLion + 31,7% PyCharm). A **Argentina** mantém um nicho próprio de Geany (22,8%), incomum em outros países. **México** e **Chile** concentram-
-se mais em VS Code (82% e 84%).

Os perfis se acentuam no PDA, onde sobem ao palco os melhores de cada país. No PDA 2026 (12 equipes brasileiras), **50% das equipes do Brasil usaram Vim**, contra 0% no México (6 equipes) e 0% na Bolívia (43º colocado, 1 equipe). A Argentina manteve o nicho de Geany (28,6% das suas 7 equipes). É a evidência mais clara de cultura local de ferramentas atuando, num campo onde o *hardware* é o mesmo para todos.

5. Editores estagnados e ausências surpreendentes

Olhar a adoção pelo prisma da maturidade do projeto traz uma surpresa. O **Code::Blocks** [8] ficou cinco anos sem lançamento estável, da versão 20.03 (março de 2020) à 25.03 (maio de 2025); como nossa imagem é baseada no Ubuntu 24.04, as máquinas ainda trazem a 20.03, então os times competiram com um editor já bem datado. Ainda assim aparece em 11% a 21% das equipes nas fases brasileiras (Regional e Final) e atinge **54,5% em Cuba e 42,1% na Bolívia** na Final. A leitura mais plausível é cultural: disciplinas introdutórias de C/C++ formaram gerações inteiras em Code::Blocks, e o hábito sobrevive a longos hiatos do projeto. Já o Geany [9], comumente colocado no mesmo balaio de "editores antigos", segue em manutenção ativa, com a versão 2.0 em 2023 e a 2.1 em 2025; o nicho argentino de 22,8% na Final é coerente com leveza somada a desenvolvimento corrente.

Pelo lado oposto chama atenção o **Emacs** [4]: projeto muito ativo (a versão 30 saiu em 2025) e mesmo a 29 do Ubuntu 24.04 já traz *tree-sitter* e cliente LSP, mas sua adoção nunca passou de 5% (0,2% a 4,5%). Falta-lhe o que faz do VS Code o vencedor: setup mínimo, *debugger* gráfico embutido e extensões pré-empacotadas para C++ competitivo; configurar o Emacs a esse nível exige tempo que poucos times investem.

6. Considerações finais

A leitura central é de associação, não de causa: o caminho mais plausível é o inverso, e os editores leves aparecem em cerca de uma de cada quatro equipes do Top-30 nos seis eventos. **Para treinadores**, a escolha modal de quem chega ao topo ainda é o VS Code, mas vale incluir um editor leve (Vim, gedit ou Geany) no repertório de treino. **Para times brasileiros**, o perfil "Vim no topo" é uma marca local: 21% na Final e 50% no PDA.

Um ponto a acompanhar é o **CLion**, cuja adoção subiu entre os ciclos (8,8% para 12,5% na Regional e 12,1% para 16,4% na Final), na contramão do "vale dos IDEs". O movimento coincide com a chegada do *Full Line Code Completion*, a complementação de código por IA local embarcada nas IDEs da JetBrains [11]. Fica a pergunta para os próximos ciclos: a complementação por IA atrairá mais times ao CLion, ou, numa prova cronometrada, atrapalha mais do que ajuda? A própria adoção responderá.

Referências

[1] Davi Antônio da Silva Santos and Bruno César Ribas. Maratona linux a tale of upgrading from ubuntu 20.04 to 22.04. Disponível em: https://arxiv.org/abs/2510.15263, 2025.
[2] Cassio P de Campos and Carlos E Ferreira. Boca: um sistema de apoio a competições de programação. In *Workshop de Educação em Computação*. Sociedade Brasileira de Computação, 2004. Disponível em: https://research.tue.nl/files/134832343/decampos2004d.pdf. Acesso em: maio 2026.
[3] Geany Project. Geany. Disponível em: https://www.geany.org. Acesso em: maio 2026.
[4] GNU Project. Gnu emacs. Disponível em: https://www.gnu.org/software/emacs/. Acesso em: maio 2026.
[5] JetBrains. Intellij idea, clion e pycharm. Disponível em: https://www.jetbrains.com. Acesso em: maio 2026.
[6] Microsoft. Visual studio code. Disponível em: https://code.visualstudio.com. Acesso em: maio 2026.
[7] Wall Berg Morais and Bruno Cesar Ribas. Maratona-linux: um ambiente para a maratona de programação. *Anais do Computer on the Beach*, pages 416–426, 2019. Disponível em: https://www.researchgate.net/publication/332247662_Maratona-Linux_um_ambiente_para_a_Maratona_de_Programacao. Acesso em: maio 2026.
[8] Anton Semenkin et al. Full line code completion: Bringing AI to desktop. Disponível em: https://arxiv.org/abs/2405.08704, 2024. Acesso em: maio 2026.
[9] The Code::Blocks Team. Code::blocks. Disponível em: https://www.codeblocks.org. Acesso em: maio 2026.
[10] The GNOME Project. gedit. Disponível em: https://gedit-text-editor.org/. Acesso em: maio 2026.
[11] Vim Project. Vim — the ubiquitous text editor. Disponível em: https://www.vim.org. Acesso em: maio 2026.

INTRODUÇÃO À OTIMIZAÇÃO POR DIVISÃO E CONQUISTA

Pedro V. Sousa da Silva, UFPR
André L. P. Guedes, UFPR

Resumo
Este artigo apresenta a otimização por divisão e conquista (Divide and Conquer Optimization), uma técnica geralmente usada para acelerar algoritmos de programação dinâmica. Sua aplicação é possível em problemas cujas funções de custo satisfazem propriedades de monotonicidade. O trabalho apresenta a ideia desta técnica, uma formalização, uma implementação generalizável e exemplos de aplicação. Este artigo é fortemente inspirado no texto Algoritmos em matrizes monótonas e Monge convexas, de Victor Sena Molero [7].

1. Problema inicial

Imagine o cenário onde temos uma matriz A de dimensões $N \times M$ e o objetivo seja determinar, para cada linha, a posição de seu menor elemento. Sem nenhuma condição a mais, não seria possível executar essa tarefa em tempo melhor que $\Omega(NM)$.

Suponha agora que a matriz não seja fornecida explicitamente, mas sim por meio de uma função

$$C(i, j) = A[i][j], \text{ para } 1 \leq i \leq N \text{ e } 1 \leq j \leq M$$

que pode ser avaliada em tempo O(1), assim sua leitura e armazenamento não impactam na complexidade do algoritmo. Além disso, suponha que C satisfaça uma propriedade de monotonicidade em um de seus índices. Nessas condições, é possível acelerar essa tarefa para $O((N + M) \log (N))$ por meio da técnica de otimização por divisão e conquista.

Uma descrição mais formal é dada a seguir.

Definição 1 (Vetor de índices de mínimos das linhas): Seja A uma matriz de dimensões $N \times M$. Definimos o vetor opt como,

$$\text{opt}[i] = \min \{j \mid A[i][j] \leq A[i][j'], 1 \leq j' \leq M\}$$

ou seja, $opt[i]$ guarda a posição da coluna que atinge o valor mínimo na linha i. Note que, em caso de empate, escolhe-se a posição mais à esquerda.

Definição 2 (Monotonicidade): Dizemos que o vetor opt é monotônico se $opt[i] \leq opt[i+1]$ para $1 \leq i < N$.

Desse modo, quando a monotonicidade é satisfeita para o vetor de índices de mínimos das linhas (opt) de uma matriz A e esta pode ser computada implicitamente, a otimização por divisão e conquista permite calcular todo o vetor opt em tempo $O((N + M) \log (N))$.

2. Ideia do algoritmo

Considere que desejamos computar o vetor opt para linhas com índices no intervalo $[l_i, r_i]$ e que as possíveis colunas ótimas atualmente estão no intervalo $[l_j, r_j]$.

Inicialmente, seja i a linha no meio do intervalo $[l_i, r_i]$, para esta linha encontramos o valor $opt[i]$ de maneira ingênua (busca linear). (Veja a Figura 1a)

Em seguida, o problema é dividido em dois subproblemas: um para linhas anteriores a i e outro para linhas posteriores a i. Sabemos pela monotonicidade de opt, que para todo $i' < i$ vale $opt[i'] \leq opt[i]$, ou seja, para todas as linhas entre $[l_i, i - 1]$ podemos limitar a busca da coluna ótima pela direita por $opt[i]$. De maneira similar, como para todo $i'' > i$ vale $opt[i''] \geq opt[i]$, podemos utilizar $opt[i]$ para restringir a busca pela esquerda para todas as linhas entre $[i + 1, r_i]$. Assim, esses dois subproblemas podem ser resolvidos de maneira recursiva até que todas as linhas tenham sido processadas. (Veja as Figuras 1b e 1c).

Em resumo, uma vez resolvida a linha do meio, podemos utilizar sua resposta para resolver recursivamente suas duas metades, com a vantagem de reduzir o intervalo de busca da coluna ótima. O Código 1 utiliza essa ideia.

```cpp
void solve(int li, int ri, int lj, int rj) {
    if (lj > rj) return;

    int i = (li + ri) / 2;
    opt[i] = lj;

    for (int j = lj; j <= rj; j++) {
        if (C(i, j) <= C(i, opt[i]))
            opt[i] = j;
    }

    solve(li, i - 1, lj, opt[i]);
    solve(i + 1, ri, opt[i], rj);
}
```

CÓDIGO 1: IMPLEMENTAÇÃO DA OTIMIZAÇÃO POR DIVISÃO E CONQUISTA.

A chamada inicial do algoritmo é **solve(0, n − 1, 0, m − 1)**, assumindo que a implementação utiliza índices iniciando em zero.

Para análise de complexidade, seja $T(N, M)$ a função de complexidade de tempo desse algoritmo. No pior caso

$$T(N, M) = 2 \cdot T\left(\tfrac{N}{2}, \tfrac{M}{2}\right) + \mathcal{O}(M) \text{ que é } \mathcal{O}((N + M)\log(N)).$$

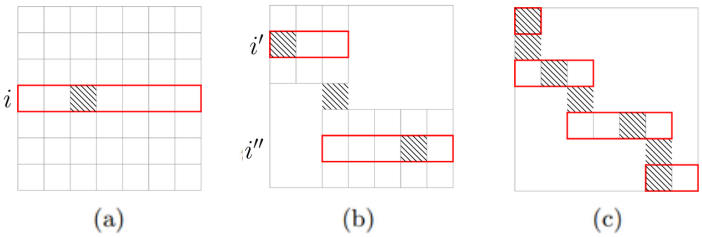


FIGURA 1: EXECUÇÃO DO ALGORITMO. A CADA NÍVEL DA CHAMADA DA FUNÇÃO, AS CÉLULAS EM VERMELHO SÃO PERCORRIDAS LINEARMENTE E OS MÍNIMOS SÃO HACHURADOS. FIGURA ADAPTADA DE MOLERO [7].

3. Problema de exemplo

Considere o seguinte problema de exemplo, adaptado do problema *Kimchi* (BOJ 11001) [4].

Problema 1: Dados dois vetores T e V, ambos de tamanho N, tais que $T_i \geq T_{i+1}$, e uma função $C(i, j) = (j - i) \cdot T_j + V_i$. Encontre o valor de $\min \{ C(i, j) \mid 1 \leq i, j \leq N \}$. Considere que $0 \leq T_i, V_i \leq 10^9$, $0 \leq N \leq 10^5$, e que todos os valores são inteiros.

Uma solução ingênua que verifica todos os pares (i, j), tem complexidade $O(N^2)$, o que é inviável para esses limites.

Mostraremos que é possível aplicar a técnica apresentada anteriormente, resultando numa solução $O(N \log N)$.

Primeiramente definimos

$$\text{opt}[i] = \min \{j \mid C(i, j) \leq C(i, j') \text{ e } 1 \leq j' \leq N\}$$

Assim, a resposta do problema será

$$\min \{C(i, \text{opt}[i]) \mid 1 \leq i \leq N\}$$

Note que se provarmos que $opt[i] \leq opt[i+1]$ basta utilizar a função C definida neste problema no Código 1 e o problema estará resolvido.

4. Demonstração

Para realizar essa demonstração utilizaremos a *Propriedade de Monge* (também conhecida como *Desigualdade Quadrangular*), enunciada a seguir.

Teorema 1 (Propriedade de Monge): Seja C uma função de custo e opt o vetor de índices de mínimos das linhas como definido anteriormente. Se, para quaisquer inteiros $a \leq b \leq c \leq d$, vale $C(a, d) + C(b, c) \leq C(a, c) + C(b, d)$, então $opt[i] \leq opt[i+1]$.

A demonstração pode ser encontrada em Molero [7]. Note que a volta nem sempre é verdadeira e nesses casos é preciso utilizar argumentos mais criativos para demonstrar a monotonicidade.

No nosso problema, basta verificar que $C(i, j) = (j - i)T_j + V_i$ satisfaz a propriedade de Monge. De fato,

$$\begin{aligned}
C(a,d) + C(b,c) &\leq C(a,c) + C(b,d) \\
(d-a)T_d + V_a + (c-b)T_c + V_b &\leq (c-a)T_c + V_a \\
&\quad + (d-b)T_d + V_b \\
(d-a)T_d + (c-b)T_c &\leq (c-a)T_c \\
&\quad + (d-b)T_d \\
0 &\leq (c-a-c+b)T_c \\
&\quad + (d-b-d+a)T_d \\
0 &\leq (b-a)T_c + (a-b)T_d \\
0 &\leq (b-a)(T_c - T_d).
\end{aligned}$$

Como $b \geq a$, temos $b - a \geq 0$. Além disso, como $c \leq d$ e o enunciado garante que $T_i \geq T_{i+1}$, segue que $T_c \geq T_d$. Logo, $(b - a)(T_c - T_d) \geq 0$, concluindo que C satisfaz a propriedade de Monge.

Observe ainda que a desigualdade quadrangular depende apenas da parte da função que envolve simultaneamente os dois índices. Assim, decompondo

$$C(i, j) = x(i, j) + y(j) + z(i)$$

os termos $y(j)$ e $z(i)$ são cancelados na desigualdade de Monge, como acontece com V_i nesse exemplo. Nesse problema, poderíamos decompor

$$C(i,j) = -iT_j + jT_j + V_i$$

e portanto, bastaria verificar que a função $x(i, j) = -iT_j$ satisfaz a Propriedade de Monge.

5. Programação dinâmica

Considere a seguinte recorrência de programação dinâmica.

Sejam N e K inteiros tais que $1 \le K \le N$ e seja $C(i, j)$, $1 \le i, j \le N$, uma função de custo. Definimos

$$E[k,i] = \begin{cases} \min\limits_{i \le j \le N} \{C(i,j) + E[k-1, j+1]\}, & \text{se } k > 1, \\ C(i,N), & \text{se } k = 1 \end{cases}$$

para $1 \le k \le K$ e $1 \le i \le N$.

Uma implementação direta dessa recorrência possui complexidade $O(KN^2)$, uma vez que, para cada estado (k, i), é necessário percorrer todos os possíveis j.

Suponha agora que a função $C(i, j)$ satisfaça a propriedade de Monge ou, mais genericamente, que o vetor de índices de mínimos das linhas seja monotônico. Nesse caso, a otimização por divisão e conquista pode ser aplicada independentemente para cada $k \ge 2$.

De fato, para um valor fixo de k, basta definir a função

$$C'(i,j) = C(i,j) + E[k-1, j+1]$$

e como o termo $E[k - 1, j + 1]$, para um k fixo, depende apenas de j, ele não interfere na propriedade de Monge nem na monotonicidade do vetor de índices de mínimos das linhas. Assim, para cada k ($2 \le k \le K$), podemos calcular

$$\min_{1 \le i,j \le N} C'(i,j)$$

em tempo $O(N \log(N))$ utilizando a otimização por divisão e conquista.

Consequentemente, toda a programação dinâmica é resolvida em $O(KN \log(N))$. Uma demonstração completa desse resultado também pode ser encontrada em Molero [7].

6. Problema de exemplo

Considere o seguinte problema de exemplo, adaptado do problema *Subarray Squares* (CSES 2086) [5].

Problema 2: Dado um vetor V de tamanho N e um inteiro K, particione V em exatamente K subvetores contíguos. O custo de um subvetor é definido como o quadrado da soma de seus elementos.

O objetivo é *minimizar* o custo total da partição. Considere que $1 \le V_i \le 10^5$, $1 \le K \le N \le 3000$, e que todos os valores são inteiros.

Seja $C(i, j) = (\sum_{x=i}^{j} V_x)^2$, a recorrência resolve o problema, isto porque podemos considerar $E[k, i]$:= menor custo de particionar o subvetor $V[i...N]$ em k partes. Assim, para solucionar o problema, basta verificar que $C(i, j)$ satisfaz a condição de monge e substituir no Código 1 com adição do termo da programação dinâmica.

7. Conclusão

Além dos exemplos apresentados neste artigo, alguns exemplos interessantes que utilizam essa otimização incluem *Money for Nothing (World Finals 2017)* [3] e sua variação *Analyzing Contracts* [1]. Também o problema *Yet Another Minimization Problem* [2] que combina essa técnica com o "Algoritmo de Mo", utilizado para calcular eficientemente a função de custo [6].

Referências

[1] Agustín Santiago Gutiérrez. Analyzing contracts. Disponível em: https://codeforces.com/gym/104736/problem/A, 2023. Acesso em: 06 jul. 2026.
[2] Codeforces. Yet another minimization problem. Disponível em: https://codeforces.com/contest/868/problem/F, 2017. Acesso em: 06 jul. 2026.
[3] ICPC World Finals 2017. Money for nothing. Disponível em: https://codeforces.com/gym/101471, 2017. Acesso em: 06 jul. 2026.
[4] Jaehyun Koo. Boj 11001. Disponível em: https://web.archive.org/web/20260419163141/https://www.acmicpc.net/problem/11001, 2015. Problema do Baekjoon Online Judge. Acesso em: 06 jul. 2026.
[5] Antti Laaksonen. Subarray squares. Disponível em: https://cses.fi/problemset/task/2086, 2026. Problema do CSES Problem Set. Acesso em: 06 jul. 2026.
[6] Mikhail Tikhomirov. Codeforces 438 analysis. Disponível em: https://codeforces.com/blog/entry/55046, 2017. Acesso em: 06 jul. 2026.
[7] Victor Sena Molero. Algoritmos em matrizes monótonas e monge convexas. Disponível em: https://linux.ime.usp.br/~victorsenam/mac0499/monografia/main.pdf, 2021. Acesso em: 06 jul. 2026.



DIAMANTE

INCOGNIA

A Maratona de Programação faz parte da história da Incognia desde a sua fundação. Nosso time reúne profissionais que participaram de finais nacionais e mundiais e que trouxeram dessa experiência o gosto por resolver problemas difíceis, o trabalho em equipe e a busca por soluções eficientes.

Na Incognia, aplicamos esse conhecimento à prevenção de fraudes digitais. Desenvolvemos tecnologia que combina inteligência de dispositivos e localização para ajudar empresas a proteger contas e transações, identificando situações de risco sem criar barreiras desnecessárias para usuários legítimos.

Lidar com grandes volumes de dados, reconhecer padrões e responder a novas técnicas de fraude são alguns dos desafios da nossa engenharia. São problemas que exigem investigação, criatividade e rigor técnico — capacidades que os maratonistas exercitam a cada competição e que têm aplicações importantes fora dela.

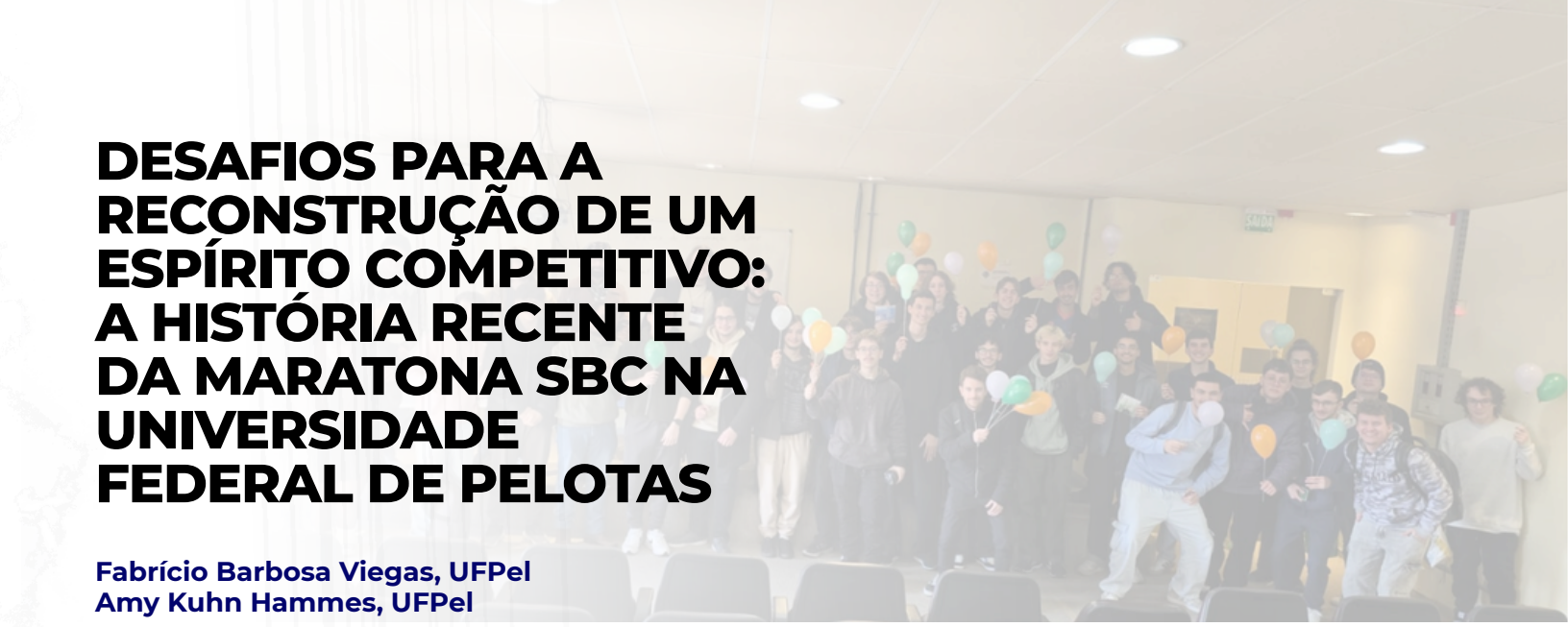
É por reconhecer o valor dessa experiência que patrocinamos a Maratona há vários anos. Em 2026, seguimos com os competidores na final em Uberlândia, apoiando um espaço de aprendizado, troca e formação que também ajudou a construir o nosso time. Aos participantes, desejamos uma excelente final!



FINAL NACIONAL 2025



A final brasileira de 2025 ocorreu no bairro da Liberdade em São Paulo, de 6 a 9 de novembro, na FECAP. A Profa. Lucy Tabuti, juntamente com os Profs. Eduardo Savino e Victor Rosseti foram os diretores da final. Os patrocinadores foram Neospace (premium); Under (infra-estrutura), Incognia, VTex, Nubank e Quinto Andar (diamante); Instituto Criativo (ouro); Alura e Mov Tech 2030 (prata); Deco e Enter (startups); Alphabot, Google, Stone, Mendelics and Valor (bronze) e contou com a parceria da CAPES, CNPa, Nikkey, Red Bull, Meruru, Voz, Novvacore, Smit and GameFY. Participaram 66 equipes na final.



DESAFIOS PARA A RECONSTRUÇÃO DE UM ESPÍRITO COMPETITIVO: A HISTÓRIA RECENTE DA MARATONA SBC NA UNIVERSIDADE FEDERAL DE PELOTAS

Fabrício Barbosa Viegas, UFPel
Amy Kuhn Hammes, UFPel

Resumo

Este artigo relata como uma universidade que permaneceu mais de dez anos sem equipes na Maratona SBC reconstruiu sua cultura de programação competitiva, passando de uma única equipe em 2023 para uma competição local com quinze equipes em 2026. Abordamos os desafios de fomentar um "espírito competitivo" em um ambiente acadêmico, detalhando desde a formação da nossa primeira equipe em 2023 até a conquista do primeiro lugar em nossa sede de prova, e a construção de nossa própria competição local. Compartilhamos a experiência de fundar essa cultura competitiva do zero, os bastidores de nossa preparação e a visão da maratona como um esporte intelectual capaz de impulsionar tanto a pesquisa quanto o desenvolvimento profissional. Espera-se que esta jornada inspire outras universidades a consolidarem sua cultura de programação competitiva.

1. Prelúdio

A Universidade Federal de Pelotas tem dois cursos na área de computação: Ciência da Computação, fundado em 1994, e Engenharia de Computação, criado em 2010. Poucos sabem que a relação da UFPel com a Maratona SBC de Programação [4] tem início bem antes disso, ela data de 2002 [5].

Naquele ano, os times Maragatos e Tchê, orientados pelo Prof. Gil Medeiros, marcaram a estreia da universidade na competição. Após um breve hiato, o cenário voltou a se animar em 2008, quando um novo grupo de estudantes, apoiados pelo Prof. Ricardo Matsumura, passou a organizar treinos e garantir presença regular nas edições seguintes. O auge veio em 2010 e 2012, quando Pelotas sediou etapas regionais da Maratona, com apoio do Facebook em um cenário aquecido que movimentava 5 universidades da região.

Após 2012, porém, o cenário foi se apagando. Não houve um evento definitivo, nem uma ruptura clara, apenas uma ausência que foi se tornando permanente. Por mais de uma década, a UFPel ficou fora das listas de inscrição, sem treinos regulares, sem times, sem cultura estabelecida. A chama simplesmente foi apagada.

Era esse o cenário da nossa instituição em 2023, quando ingressamos no curso de Ciência da Computação.

2. O recomeço de tudo

Para entender esta história, é preciso saber que a nossa relação com programação competitiva começa muito antes da graduação. Ambos somos apaixonados pela área desde a adolescência [2], graças a iniciativas como a OBI, o tipo de paixão que cresce resolvendo problemas em juízes online nas madrugadas, que dificulta imaginar a vida universitária sem alguma competição, que faz com que a primeira pergunta ao chegar numa nova universidade seja: “tem maratona aqui?”. Na UFPel, a resposta era não. Mas isso, como descobriríamos logo, era apenas um ponto de partida.

O momento decisivo ocorreu logo na primeira semana de aula, numa disciplina de Algoritmos e Programação, uma cadeira introdutória do curso. O professor que a ministrava ao apresentar o plano pedagógico, mencionou uma categoria reservada para prêmios e distinções. Naquele instante, levantamos as mãos. “Pode ser uma maratona de programação?”, mal sabíamos que seria o início de uma parceria que se traduziria em algo expressivo anos depois.

Logo após a aula já estávamos organizando o que seria o primeiro time da universidade na competição em muito tempo. Procuramos o professor Guilherme Netto, que ministrava a emblemática disciplina, conseguimos seu apoio como nosso primeiro coach e começamos a recrutar colegas. Foi um esforço de guerrilha, feito nos corredores, nas conversas entre aulas, tentando convencer quem aparecia pela frente de como tudo aquilo poderia ser divertido, com um sucesso variável, é justo admitir.

Assim seria por muito tempo, seríamos praticamente os únicos sustentando o projeto. Dois estudantes, acompanhados de um terceiro membro que por vezes nem sabia o que estava acontecendo ali, representando sozinhos uma universidade, carregando nas costas a vontade de ver crescer algo que ainda não existia. Havia dias em que nos perguntávamos se conseguiríamos despertar o interesse genuíno de mais alguém. A resposta, por um bom tempo, não era animadora. Mas o interesse pelo que fazíamos nunca diminuiu, e foi isso que nos manteve em movimento.

3. A Estreia (2023)

Em 2023, a UFPel retornava à Maratona SBC de Programação na super-site Porto Alegre, depois de mais de uma década de ausência. A equipe era pequena, quatro estudantes e o professor Netto como coach, não carregávamos grandes expectativas de resultado. Carregávamos, apenas entusiasmo, seguramente é possível afirmar que éramos os mais barulhentos naquele auditório em Porto Alegre.



FIGURA 1: NOSSO PRIMEIRO TIME, EM ORDEM: PROF. GUILHERME NETTO, FABRÍCIO VIEGAS, AMY KUHN, JOLENE JUSTIN E PEDRO JACINTO (RESERVA).

Aquela primeira edição foi, acima de tudo, um exercício de aprendizado. Aprendemos a dinâmica da competição, o formato, a lógica de divisão de tarefas e a pressão de trabalhar em equipe. Levamos uma surra, saímos de lá com aprendizados que nenhum treino improvisado havia nos dado e com a certeza de que queríamos continuar isso de forma mais estruturada.

4. Ano dois (2024)

No ano seguinte, voltaríamos a Porto Alegre para competir mais uma vez, com uma nova formação e sob o apoio do nosso novo coach, Prof. Rafael Torchelsen, que, com os problemas de agenda do Prof. Netto, se tornaria nosso coach definitivo e havia se mostrado muito animado com a iniciativa.



FIGURA 2: TIME 2024/2025, ADICIONADOS: PROF. RAFAEL TORCHELSEN E MURILO SALEM. PEDRO COMO MELÂNCIA.

A edição de 2024 trouxe um desafio diferente, nossos resultados já eram melhores com treinos mais estruturados, mas ficava claro que sem sede local, participar significava viajar até Porto Alegre. Para nós, isso não era impedimento, iríamos de qualquer forma mesmo que endividados. Mas sabíamos que, para outros estudantes da UFPel, o custo do deslocamento era uma barreira real.

Aquela edição consolidou o projeto, mas também deixou evidente um problema estrutural que não dependia de esforço ou talento para resolver. Enquanto competir exigisse viajar, seria muito mais difícil ampliar a participação da UFPel. A limitação não era de vontade, era distância e dinheiro.

5. Campeões em Casa (2025)

A edição de 2025 mudou a equação de forma que não havíamos antecipado. A UCPel, universidade católica da cidade, foi escolhida como sede da fase regional em Pelotas, pela primeira vez, não havia necessidade de viajar. Mais estudantes da UFPel podiam ser inscritos, junto de nosso coach, agora fixo, convidamos colegas a nos encontrarem num final de semana, onde aplicamos simulados e explicamos o formato da prova, também ministramos um curso sobre programação competitiva na semana acadêmica dos cursos de computação (SACOMP) daquele ano [1], e levamos à competição mais times do que em qualquer edição anterior.



FIGURA 3: CURSO DE PROGRAMAÇÃO COMPETITIVA MINISTRADO NA SACOMP25.

No dia 3 de setembro, ao longo de cinco horas de prova, mostramos o que três anos de esforço contínuo haviam construído. Ao final, conquistamos o primeiro lugar entre os times que competiram na sede da UCPel e o 13º lugar no super-site Pelotas–Porto Alegre, resultado que não garantiu classificação para a fase nacional, mas que reconheceu, de forma inequívoca, a trajetória percorrida.



FIGURA 4: ALUNOS DA UFPEL NA MARATONA DE 2025 NA UCPEL

A conquista chegou ao conhecimento da universidade por outros caminhos também. Naquele mesmo ano, fomos entrevistados por um projeto do curso de Jornalismo da nossa universidade [3]. Sobre esse convite, sentimos que algo havia se consolidado, não apenas um resultado, mas uma narrativa. A UFPel estava de volta, e havia uma história para contar.

6. De Um Time a Quinze (2026)

Se as competições oficiais marcaram a nossa trajetória como equipe, a SACOMP 2026 representou uma ambição diferente: semear algo além de nós mesmos.

A convite de um membro da organização da SACOMP e com a ajuda dele, assumimos junto de colegas sem nenhuma experiência em competições desse tipo, a organização da 1ª Maratona de Programação da SACOMP, realizada em 26 de junho de 2026. Inspirada no formato da Maratona SBC, a competição aconteceu presencialmente, utilizando o Beecrowd como juiz, e reuniu 15 equipes de duas universidades.

Chegar a esse número não foi trivial. Nas semanas que antecederam o evento, os professores abriram espaço em suas disciplinas de Algoritmos e Estruturas de Dados para simulados, apresentação do formato e convite a novos participantes. Foi uma mobilização que conhecemos bem: parte institucional, parte artesanal, a mesma lógica de corredor que havíamos praticado desde o começo.

O dia da competição foi intenso, desafiador e, acima de tudo, muito divertido. A equipe campeã levou para casa uma premiação em dinheiro, e conseguimos garantir que todos os participantes recebessem alguma forma de reconhecimento, das premiações mais expressivas às lembranças simbólicas. Para nós, porém, o número que mais importava estava na lista de inscrições.

Quinze times. Quando, há pouco tempo, éramos apenas um.

7. Considerações finais: O que fica

A edição de 2026 da Maratona SBC será nossa última como competidores. Encerraremos o ciclo da graduação e, com ele, nossa participação direta na competição, dessa vez novamente com um terceiro membro novo, na esperança de passar o bastão para a nova geração. Queremos ir bem, claro. Quem passa anos construindo algo quer se despedir à altura. O crescimento do cenário competitivo na região nos dá razões concretas para otimismo: mais times, mais universidades, uma maratona local estabelecida, estudantes que antes não conheciam o formato e hoje já querem organizar a próxima edição.



FIGURA 5: TIMES NA I MARATONA SACOMP.

Mas o que mais nos importa não é o placar da nossa despedida. É a resposta para uma pergunta que, no começo, não sabíamos se teria resposta: é possível reconstruir do zero uma cultura competitiva numa universidade?

Sim. É possível. Demanda obstinação, parceiros dispostos, professores que abram espaço e, sobretudo, uma paixão pelo que se faz que resista às temporadas em que parece que mais ninguém está interessado. Mas é possível.

Quando começamos, éramos dois alunos tentando convencer colegas no corredor. Três anos depois, quinze equipes competiam em uma maratona organizada pelos próprios estudantes.

Quando levantamos as mãos naquela aula de A&P, não sabíamos exatamente o que estávamos iniciando. Sabíamos apenas que queríamos competir e que havia outros que poderiam se apaixonar por isso também, se tivessem a oportunidade.

Esperamos que esta jornada inspire outras universidades, professores e alunos a fazerem o mesmo.

8. Agradecimentos

Os autores agradecem ao Prof. Guilherme Netto, primeiro coach da equipe, cujo entusiasmo inicial tornou tudo possível; ao Prof. Rafael Torchelsen, que tem se demonstrado um amigo e parceiro
fiel nessa empreitada, e à Prof.ª Brenda Salenave Santana, pelo apoio que fez a SACOMP acontecer; e a Diogo Krüger, parceiro na organização do evento.

Referências

[1] Fabrício Barbosa Viegas and Amy Kuhn Hammes. Material do curso de programação competitiva UFPel. Disponível em: https://github.com/fabriciobarbosaviegas/Curso-de-programa-o-competitiva-SACOMP-2025, 2025. Acesso em: 28 jun. 2026.
[2] Redação Blog do Juares. Jovem camaquense se classifica para fase estadual da olimpíada brasileira de informática. Disponível em: https://blogdojuares.com.br/noticia/69100/jovem-camaquense-se-classifica-para-fase-estadual-da-olimpiada-brasileira-de-informatica.html. Acesso em: 28 jun. 2026.
[3] Drops em Pauta UFPel. Maratona de programação na UFPel. Disponível em: https://www.instagram.com/p/DQGFqRfgNh8/, 2025. Acesso em: 28 jun. 2026.
[4] Sociedade Brasileira de Computação. Maratona SBC de programação. Disponível em: https://maratona.sbc.org.br. Acesso em: 28 jun. 2026.
[5] Universidade Federal de Pelotas. Histórico da maratona na UFPel. Disponível em: https://wp.ufpel.edu.br/maratona/historico-da-maratona-na-ufpel/. Acesso em: 28 jun. 2026.

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);
}

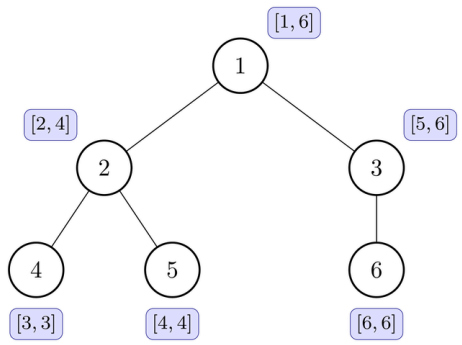
// (ii)
ll dfs(int u, int p, int d = 0) {
    ll subtree = add[d];

    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.



DIAMANTE

Apoiar um projeto como a Maratona de Programação tem tudo a ver com o que o Nubank se propõe a fazer: desafiar o status quo e questionar formas convencionais de fazer as coisas. A competição estimula justamente esse pensamento, colocando estudantes diante de problemas complexos e incentivando soluções criativas, colaborativas e fora do óbvio.

Para o Nubank, estar próximo desse ecossistema é também uma forma de apoiar quem está pensando diferente e ajudando a construir o futuro da tecnologia. Mais do que uma iniciativa de atração de talentos, é uma oportunidade de contribuir para o desenvolvimento de uma comunidade que compartilha valores muito próximos aos nossos.

Acreditamos que transformar indústrias começa por pessoas dispostas a perguntar “por que não?” e buscar novas respostas. Apoiar a Maratona é uma forma concreta de fomentar esse espírito e levar adiante a mesma mentalidade de desafiar o convencional que está no centro do Nubank.

MARATONAS PELO BRASIL AFORA

Ao longo dos anos, diferentes regiões, estados e instituições de ensino desenvolveram suas próprias competições de programação, inspiradas pelo movimento da Maratona SBC. Essas iniciativas surgiram tanto para aumentar o interesse de estudantes, professores e empresas pela programação competitiva quanto para elevar o nível dos competidores da região nas etapas nacionais. Cada competição mantém sua própria identidade, incorporando, em alguns casos, elementos da cultura e das características locais aos problemas e à organização da prova. A seguir, apresentamos algumas das principais competições regionais realizadas no Brasil.



Maratona Interfatecs

Criada em 2012, com sua primeira edição sediada na FATEC Sorocaba, a Maratona INTERFATECS chega à sua 15ª edição em 2026. A competição reúne as Faculdades de Tecnologia (FATECs) do estado de São Paulo e é dividida em duas fases: uma fase inicial, realizada internamente em cada unidade, e uma fase final, que reúne as melhores equipes classificadas de cada FATEC.



Maratona Mineira

Criada em 2012, no estado de Minas Gerais, considerado um dos berços da Maratona de Programação no Brasil, a Maratona Mineira teve como primeira sede a cidade de Uberlândia. Com 13 edições realizadas, é uma das mais antigas entre as competições regionais apresentadas nesta seção. Sua edição mais recente foi realizada na Universidade Federal de Itajubá (UNIFEI), em Itabira, e contou com 58 equipes de 28 instituições de ensino diferentes. A equipe da Universidade Federal de Minas Gerais (UFMG) conquistou o título, tornando-se campeã da competição pela sétima vez. A sede mineira também conta com a Maratona Minas de Minas, iniciativa voltada a incentivar a participação de mulheres e pessoas não binárias na computação, cuja segunda edição será realizada em outubro de 2026, na Pontifícia Universidade Católica de Minas Gerais (PUC-MG).



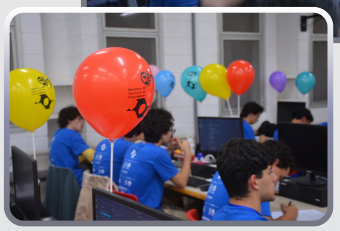
Maratona do Cerrado

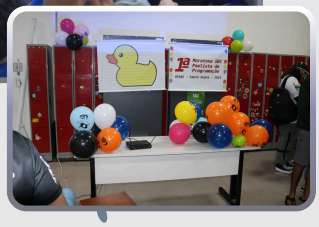
Criada em 2025, a Maratona de Programação do Cerrado é uma competição que engloba instituições de Minas Gerais, Goiás e do Distrito Federal. A primeira edição foi realizada na Universidade Federal de Uberlândia (UFU), enquanto a segunda teve como sede a Universidade de Brasília (UnB). A UnB é a atual bicampeã da competição, tendo conquistado o título nas duas primeiras edições.











Maratona Paulista

A Maratona SBC Paulista (MPP), idealizada em 2024 durante a primeira fase da Maratona SBC, é uma competição voltada às instituições de ensino superior do estado de São Paulo. Teve como inspiração a Maratona Mineira e a Maratona INTERFATECS. Até 2026, foram realizadas duas edições: a primeira, realizada em 2025 na Universidade Federal do ABC (UFABC), em Santo André; e a segunda, realizada na Universidade Estadual de Campinas (UNICAMP). Atualmente duas universidades foram campeãs, a UNICAMP em 2025 e a Universidade de São Paulo (USP) em 2026.

Maratona Nordestina

A Maratona Nordestina de Programação teve sua primeira edição em 2026, sendo sediada em sete locais diferentes e contando com a participação de 97 equipes de 14 instituições de ensino. A equipe da Universidade Federal de Pernambuco (UFPE) conquistou o título da primeira edição.







Além das competições regionais apresentadas acima, diversas outras maratonas são realizadas regularmente pelo Brasil. Entre elas, destacamos a Maratona do Norte de Programação e a Maratona do Sul de Programação, que uniram forças para organizar, em 2022, a Fase Zero da Maratona de Programação SBC, atualmente descontinuada. Também merece destaque a seletiva interna da MaratonUSP, grupo de extensão da Universidade de São Paulo (USP) voltado às maratonas e olimpíadas de programação, que realiza a seletiva desde 1997. A Maratona da UnB, por sua vez, chega à sua 14ª edição em 2026. A Maratona Cearense de Programação (MCP) que realizou sua primeira edição também no ano de 2026, sendo a segunda estreante da região nordeste. Essas são apenas algumas das diversas iniciativas que contribuem para o desenvolvimento da programação competitiva no país.

PUSH-RELABEL

Bernardo Amorim, UFMG
Kaio Vieira, UFMG

Resumo
O algoritmo de Push-Relabel foi proposto por Andrew Goldberg e Robert Tarjan em 1988 [4] e se propõe a resolver o problema de fluxo máximo. Neste artigo apresentamos esse algoritmo e demonstramos sua corretude e complexidade.

1. Intuição

Os algoritmos de fluxo mais populares, como os algoritmos de Edmonds-Karp [3] e de Dinitz [2], são baseados em iterativamente encontrar um caminho aumentante da fonte para o sumidouro e aumentar o fluxo que passa por esse caminho, se aproveitando do seguinte lema:

Lema 1. *Um fluxo é máximo se e somente se não existe caminho aumentante da fonte até o sumidouro no grafo residual.*

Desta forma, estes algoritmos começam em uma configuração em que nenhuma unidade de fluxo sai da fonte e aumentam essa quantidade por meio de caminhos aumentantes até que não seja mais possível encontrá-los, mantendo a invariante de que o fluxo é válido a cada iteração do algoritmo.

O algoritmo de Push-Relabel, por outro lado, começa em uma configuração em que todas as arestas saindo da fonte foram saturadas, passando o máximo de fluxo possível da fonte para seus vizinhos. Entretanto, apesar dessa atribuição não representar um fluxo válido, não existe caminho da fonte até o sumidouro no grafo residual e essa invariante é mantida a cada iteração do algoritmo, através de operações que retornam unidades de fluxo excedente de volta para a fonte, terminando assim que o fluxo se tornar válido.

2. Definições

Utilizaremos a seguinte terminologia:

- $G(V, E)$ = grafo de entrada onde $n = |V|$, $m = |E|$
- $G_f(V_f, E_f)$ = grafo residual
- $f(u, v)$ = fluxo de u para v em G_f, $f(u, v) = -f(v, u)$
- $c(u, v)$ = capacidade da aresta
- $c_f(u, v)$ = capacidade da aresta no grafo residual
- s = fonte, t = sumidouro

A configuração da rede residual mantida a cada passo do algoritmo é formalizada pelo conceito a seguir.

Definição 1 (Pré-fluxo). *Um pré-fluxo é uma função f das arestas para fluxo tal que:*
- *$f(u, v) \leq c(u, v)$ — o pré-fluxo não excede a capacidade da aresta.*
- *Para todo vértice $v \neq s$:*
  fluxo entrando em $v \geq$ fluxo saindo de v

Desta forma, um pré-fluxo é basicamente um fluxo onde não é necessário que tudo que entre em um vértice saia, o que motiva a nossa próxima definição:

Definição 2 *(Excesso). O excesso $e(v)$ de um vértice v é a diferença entre o quanto de fluxo entra no vértice e o quanto sai. Note que o excesso de um vértice que não é a fonte é sempre não negativo.*

Além do que foi definido acima, a noção de uma altura definida para cada vértice é uma ferramenta importante para guiar as operações a serem feitas.

Definição 3 *(Altura). Uma altura inteira não negativa $h(v)$ é associada a cada vértice v e as seguintes invariantes serão respeitadas durante o algoritmo:*
1. $h(s) = n$
2. $h(t) = 0$
3. *Para toda aresta $(u, v) \in G_f$ tal que $c_f(u, v) > 0$, $h(u) \leq h(v) + 1$ i.e. arestas no grafo residual sobem, permanecem retas ou descem uma unidade.*

3. Pseudocódigo

O algoritmo pode ser descrito por duas simples operações, PUSH (1) que nos permite empurrar fluxo através de uma aresta, e RELABEL (2) que nos permite mudar a altura de um vértice para tentar passar fluxo por ele no futuro.

Algoritmo 1 PUSH

função PUSH(u, v)
  $\Delta \leftarrow \min(e(u), c_f(u, v))$
  Empurre Δ unidades de fluxo por (u, v)
  retorne

Algoritmo 2 RELABEL

função RELABEL(v)
  $h(v) \leftarrow h(v) + 1$
  retorne

Para garantir as invariantes (as da altura e da não existência de caminho $s - t$ em G_f), inicia-se o algoritmo com o procedimento de inicialização em Algoritmo 3.

Algoritmo 3 INICIALIZAÇÃO

função INICIALIZAÇÃO
  # garantindo as 3 invariantes da altura
  $h(s) \leftarrow n$
  para $\forall v \neq s$ **faça**
    $h(v) \leftarrow 0$
    Empurre $c(s, v)$ unidades de fluxo por (s, v)
retorne

A altura de cada vértice é mantida com o intuito de proibir que operações de PUSH sejam feitas de maneira indiscriminada, para que não se mova fluxo em um ciclo. Por isso só faremos PUSH quando a aresta em questão "abaixar" uma unidade de altura. Ao não conseguir fazer PUSH partindo de um vértice com excesso, fazemos RELABEL neste. Desta forma, o pseudocódigo para o algoritmo completo segue em 4.

Algoritmo 4 PUSH–RELABEL

INICIALIZAÇÃO()
enquanto $\exists v \notin \{s, t\}$ tal que $e(v) > 0$ **faça**
  Escolha algum desses v's
  se $\exists (v, u) \in E_f$ tal que $c_f(v, u) > 0$ e $h(v) = h(u) + 1$
então
    PUSH(v, u)
  senão
    RELABEL(v)

A Figura 1 mostra um exemplo das operações realizadas pelo algoritmo. Após o último estado ilustrado, o algoritmo ainda realizaria três operações de RELABEL no vértice a, elevando sua altura de 1 para 4, e uma operação PUSH(a, s), devolvendo as duas unidades de excesso à fonte e encerrando sua execução.

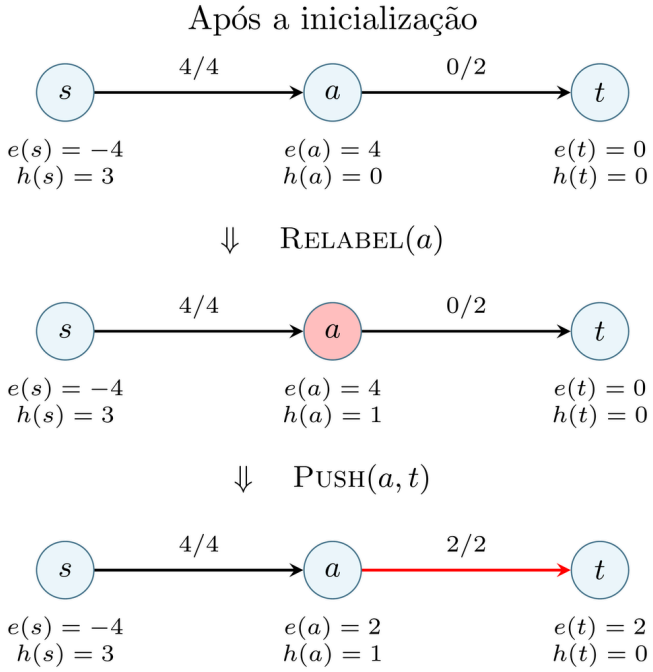


FIGURA 1: EXEMPLO DAS OPERAÇÕES RELABEL E PUSH. ABAIXO DE CADA VÉRTICE SÃO MOSTRADOS SEU EXCESSO E SUA ALTURA; EM CADA ARESTA, O RÓTULO INDICA FLUXO/CAPACIDADE.

4. Corretude

É fácil ver que as operações descritas mantêm as invariantes 1, 2 e 3. Agora demonstramos que a invariante principal é mantida:

Lema 2. *Não existe caminho $s - t$ em G_f em qualquer momento do algoritmo.*

Demonstração. Suponha a fim de contradição que esse caminho $s = v_0, v_1, \ldots, v_l = t$ existe. Então $l < |V|$. Como respeitamos a terceira invariante da altura, $h(v_i) \leq h(v_{i+1}) + 1$, desta forma temos $h(s) \leq h(t) + l < n$, o que contradiz que $h(s) = n$.

Note agora que o algoritmo só para se nenhum vértice de $V \setminus \{s, t\}$ possuir excesso. Perceba também que nessa configuração o pré-fluxo é um fluxo. Desta forma, como consequência direta dos Lemas 1 e 2, para provar que a resposta do algoritmo é ótima basta demonstrar que este termina.

5. Terminação e complexidade

Primeiro, demonstremos o seguinte lema auxiliar que, apesar de exigir alguma sofisticação técnica diz algo muito simples: se um vértice possui excesso, dado que esse excesso veio da fonte por algum caminho, deve haver alguma forma de devolvê-lo para a fonte.

Lema 3. Em qualquer momento do algoritmo, se v é tal que $e(v) > 0$, então existe caminho $v - s$ em G_f.

Demonstração. Seja S o conjunto de vértices alcançáveis por v em G_f e suponha por contradição que $s \notin S$. A escolha de S implica que, para todo u, w onde $u \in \bar{S}$, $w \in S$ temos $f(u, w) \leq 0$. Logo:

$$\begin{aligned}\sum_{w \in S} e(w) &\leq \sum_{u \in V, w \in S} f(u, w) \\ &= \sum_{u \in \overline{S}, w \in S} f(u, w) + \sum_{u, w \in S} f(u, w) \\ &= \sum_{u \in \overline{S}, w \in S} f(u, w) \leq 0 \qquad (\text{antissimetria})\end{aligned}$$

Como o excesso é não-negativo, temos que $e(v) = 0$, contradição.

Agora limitaremos o número de RELABELS que podem ser realizados durante o algoritmo.

Lema 4 (Número de Relabels). *A todo momento do algoritmo, para todo vértice v vale $h(v) \leq 2n - 1$, logo o número de* RELABELS *do algoritmo é da ordem de $O(n^2)$.*

Demonstração. O lema é trivial para s e t. Considere $v \in V \setminus \{s, t\}$. Como o algoritmo só muda as alturas de vértices ativos ($e > 0$), é suficiente provar o Lema apenas para esses vértices ativos. Se v é ativo então pelo Lema 3 existe caminho $v = v_0, v_1, \ldots, v_l = s$ em G_f onde $l \leq n - 1$. Pela terceira invariante da altura temos que $h(v_i) \leq h(v_{i+1}) + 1$. Logo temos $h(v) \leq h(s) + l \leq h(s) + (n - 1) = 2n - 1$.

Para analisar o número de PUSHES os dividiremos em duas categorias segundo a seguinte definição:

Definição 4. *Uma operação* PUSH(u, v) *é dita ser saturante se a quantidade de fluxo enviada satura a aresta* (u, v).

Lema 5. *O número de* PUSHES *saturantes do algoritmo é da ordem de $O(nm)$.*

*Demonstração***.** Entre dois PUSHES saturantes na mesma aresta (u, v) é preciso que um dos vértices sofra RELABEL pelo menos duas vezes pela condição da altura imposta no PUSH. Como cada vértice só pode ser parte de $O(n)$ RELABELS de acordo com o Lema 4, cada aresta só sofre $O(n)$ PUSHES saturantes.

Lema 6. *O número de* PUSHES *não saturantes do algoritmo é da ordem de $O(n^2m)$.*

Demonstração. Tome $\Phi = \sum_{v \text{ ativo}} h(v)$. Todo PUSH não saturante diminui Φ por pelo menos 1, uma vez que ele desativa o vértice em questão. Já uma operação de PUSH saturante faz com que Φ cresça por no máximo $2n - 1$, uma vez que ativa no máximo um vértice e cuja altura máxima é $2n - 1$ pelo Lema 4. Uma operação de relabel aumenta Φ em exatamente 1. Como Φ se inicia em zero, é fácil ver que o número de PUSHES não saturantes é limitado por $O(n^2m)$ como consequência dos Lemas 4 e 5.

Teorema 1. *A complexidade do algoritmo de Push-Relabel é da ordem de $O(n^2m)$.*

Demonstração. A complexidade do algoritmo pode ser limitada pelo número de PUSHES e RELABELS, que são da ordem de $O(n^2m)$ como visto nos Lemas 4, 5 e 6.

6. Considerações finais

Podemos observar que o algoritmo 4 não impõe nenhuma restrição sobre a ordem em que os vértices com excesso devem ser escolhidos. O estudo desses critérios é importante para que o algoritmo seja eficiente na prática. Dentre as estratégias mais comuns, escolher o vértice com excesso positivo que possuir a maior altura a cada iteração permite melhorar a complexidade assintótica para $O(n^2\sqrt{m})$ [1].

Em programação competitiva, implementações de Push-Relabel que seguem essa estratégia podem ser eficientes e enxutas, constituindo uma opção interessante para um algoritmo de fluxo máximo em materiais de referência; a implementação da KACTL é um exemplo dessa abordagem [5].

Referências

[1] Joseph Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. SIAM Journal on Computing, 18(6):1057–1086, 1989.

[2] Yefim A. Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. Soviet Mathematics Doklady, 11:1277–1280, 1970.

[3] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the Association for Computing Machinery, 19(2):248–264, 1972.

[4] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. Journal of the Association for Computing Machinery, 35(4):921–940, 1988.

[5] Simon Lindholm. Push-relabel. https://github.com/kth-competitive-programming/kactl/blob/main/content/graph/PushRelabel.h, 2015. KTH Challenge Algorithm Template Library (KACTL). Acesso em: 31 jul. 2026.

HARDWARE DAS SEDES DA MARATONA SBC: POLÍTICA, REALIDADE BRASILEIRA E PANORAMA LATINO-AMERICANO

Bruno Ribas, UnB

Resumo

A partir da telemetria de hardware das sedes da Maratona SBC (duas Regionais e duas Finais Latam), este artigo informativo e descritivo resume um achado central. Há forte disparidade regional de RAM no Brasil (média normalizada de 12,1 GB no Nordeste a 16,8 GB no Sul) e 34% dos times da Primeira Fase ainda jogam com 8 GB, faixa em que a pressão de memória durante a prova é a maior, sobretudo com IDEs pesadas como o Intellij, que paginam bem mais que o VS Code. Os dados apontam 16 GB como piso técnico adequado, mas a cobertura atual e a forte alta recente no preço da memória recomendam um cronograma gradual, não uma exigência imediata.

1. Sistema e coleta de dados

A competição se organiza em duas frentes. **A Maratona SBC de Programação** conduz a **Primeira Fase** (regional brasileira, em dezenas de sedes pelo Brasil) e a sede brasileira da **Final Latam**, a **Final Brasileira**. O **ICPC-Latam** organiza essa Final Latam, distribuída em uma sede por país latino-americano (algumas com duas, por logística), e o **PDA** (Programadores de América), final continental em sede única (cerca de 41 times). Consideramos dois **ciclos** completos, 2024–25 e 2025–26, cada um com as três etapas.

Em todas as fases o ambiente é padronizado: julgamento pelo BOCA [3] e máquinas rodando o Maratona Linux 24.04 [6, 2], baseado em Ubuntu 24.04. Sobre essa base, cada máquina grava uma telemetria simples com histórico de CPU, RAM e tempo de cada editor aberto. Placares oficiais são públicos[1].

2. Política oficial de hardware

A especificação oficial publicada[2] define um *mínimo* (multi-core a partir de 2015, Ryzen 3 ou Core i3 ou superior, 8 GB de RAM, SSD com 30 GB livres) e um *recomendado* (multi-core a partir de 2020, Ryzen 5 ou Core i5 ou superior, 16 GB de RAM). A política por fase exige hoje 8 GB na Primeira Fase, 16 GB na Final Brasileira (e ao menos 8 GB nas demais sedes da Final Latam), e 16 GB no PDA. O passo natural seria padronizar 16 GB também na Primeira Fase; os dados deste ciclo sugerem que a decisão merece cautela.

[1]https://scorelatam.naquadah.com.br/

[2]https://maratona.sbc.org.br/sobre/ambiente_computacional.html

3. Realidade brasileira: RAM e idade de CPU por região

Região	sedes	times	RAM (GB)	idade (anos)	∼8 GB (%)	16 GB (%)	>18 (%)
Norte	7	65	13,2	9,7	64,6	26,2	9,2
Nordeste	16	231	12,1	8,1	46,8	45,9	7,2
Sudeste	17	414	15,5	6,7	30,9	47,0	22,1
Sul	6	194	16,8	7,4	10,8	62,4	26,8
Centro-Oeste	5	84	19,0	8,2	40,7	44,4	14,8
Brasil	51	988	14,6	7,8	33,6	48,2	18,1

TABELA 1: REGIONAL BRASILEIRA 2025–26: RAM MÉDIA NORMALIZADA POR SEDE, IDADE MÉDIA DO PROCESSADOR E DISTRIBUIÇÃO POR FAIXA DE RAM. FAIXAS: ~8 GB (ATÉ 11 GB), 16 GB (14–18 GB), >18 GB (32 GB+); INTERMEDIÁRIAS OMITIDAS.

Consideramos só as **máquinas usadas por times** (uma por time, casada pelo log do BOCA): das 2.283 ligadas, 988 foram de fato usadas, porque muitas sedes ligam máquinas reserva. *Todas as métricas deste artigo usam esse conjunto.* A RAM por região é a *média das médias de cada sede*; a idade do processador é inferida do modelo da CPU. O processador mais comum é o Core i5 (52%), seguido do i7 (23%).

O Nordeste tem a menor RAM (12,1 GB), com o Norte logo atrás (13,2 GB); são as regiões com mais times em 8 GB (47% e 65%). **O Sul tem o melhor perfil** (16,8 GB; só 11% em 8 GB) e o **Sudeste, os processadores mais novos** (2019; 6,7 anos, ante 2016 no Norte). O Centro-Oeste aparece em 19,0 GB, mas é um *outlier* de uma única sede (Goiânia), com máquinas de 64 GB; sem ela, a região cairia para 12,3 GB, entre as mais baixas.

Ante 2024–25, a fração em 8 GB caiu de 49% para 34%: Sul (12,2 → 16,8 GB), Sudeste (11,9 → 15,5) e Centro-Oeste subiram; o Nordeste ficou em 12,1 e o Norte regrediu (16,3 → 13,2 GB), provavelmente por troca de sedes.

4. Panorama latino-americano: hardware por país

Na Final Latam de 2025, a RAM por país vai de 7,7 GB em Cuba (100% em 8 GB) a 62,5 GB em Porto Rico (Tab. 2). A maioria pôs seus times em 16 GB e alguns em 32 GB; só *Cuba e El Salvador* ficaram abaixo. A média, porém, esconde caudas, e é para isso que a tabela traz a coluna de % < 10 GB: a Argentina, mesmo com 20,0 GB de média, ainda levou 26% dos times com menos de 10 GB.

Entre os dois ciclos, a evolução é expressiva: Peru-Arequipa saltou de 7,6 para 31,0 GB e a Argentina de 7,6 para 20,0; o próprio *Brasil* subiu de 7,5 para 15,4 GB, tirando seus times do patamar de 8 GB. O contraste de escala com a Tab. 1 mostra por que a Primeira Fase é o desafio real: a Final equipa 66 times brasileiros em uma única sede, enquanto a Primeira Fase precisa equipar 988 times em 51 sedes, e é justamente nessa capilaridade que um piso maior de RAM custa caro.

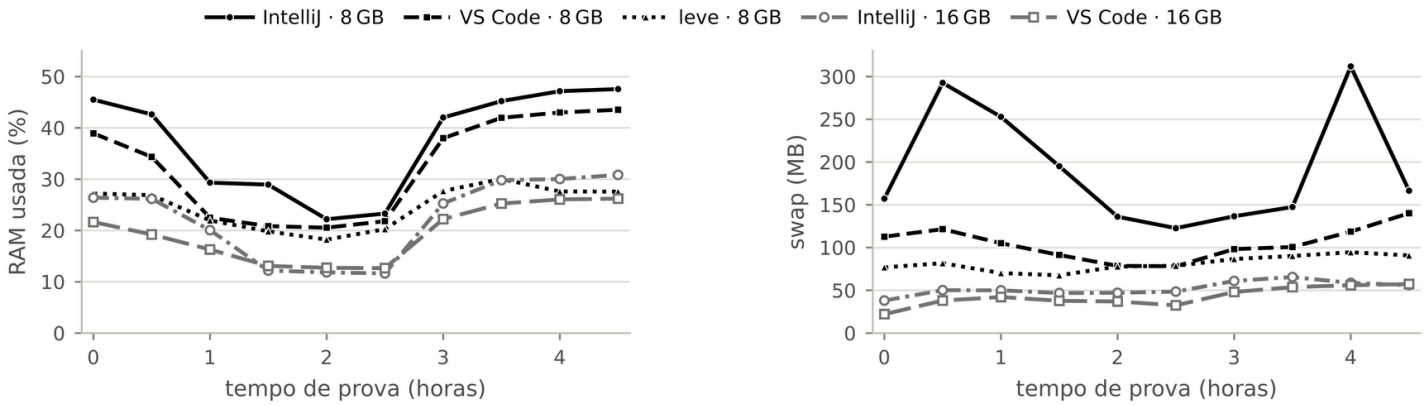


FIGURA 1: USO MÉDIO DE RAM (ESQ.) E DE SWAP (DIR.) AO LONGO DO TEMPO DE PROVA, EM JANELAS DE 30 MIN (REGIONAL 2025–26), PARA OS PERFIS DA TAB. 3.

País	times	RAM média (GB)	% < 10 GB
Cuba (CU)	22	7,7	100,0
El Salvador (EL)	15	10,3	66,7
Colômbia (CO)	41	15,0	2,4
Costa Rica (CR)	15	15,3	0,0
Venezuela (VZ)	16	15,3	0,0
Peru-Lima (PE)	22	15,3	0,0
Brasil (BR)	66	15,4	0,0
México (MX)	133	15,4	0,0
Chile (CH)	37	15,5	0,0
Argentina (AR)	57	20,0	26,3
Guatemala (GT)	20	30,3	0,0
Bolívia (BO)	76	30,8	0,0
Peru-Arequipa (PA)	9	31,0	0,0
Porto Rico (PR)	10	62,5	0,0

TABELA 2: FINAL LATAM 2025: MÉDIA DE RAM POR PAÍS E % COM MENOS DE 10 GB. ORDENADO POR RAM; ANTÍGUA, TRINIDAD E REP. DOMINICANA (AMOSTRAS PEQUENAS) OMITIDAS. PERU TEM DUAS SEDES.

5. Pressão de memória na prova

Perfil	N	mem% 0 h	mem% 4 h	swap (MB)
8 GB · IntelliJ	27	45,5	**47,2**	**189**
8 GB · VS Code	103	38,9	43,0	104
8 GB · leve	11	27,2	27,6	82
16 GB · IntelliJ	31	26,4	30,0	52
16 GB · VS Code	199	21,6	26,1	43

TABELA 3: CONSUMO DE MEMÓRIA NA PROVA POR PERFIL DE EDITOR (REGIONAL 2025–26): TIMES QUE PASSARAM A MAIOR PARTE DA PROVA (≥60% DO TEMPO) NUM EDITOR. INTELLIJ = IDEA/CLION/PYCHARM; LEVE = EXCLUSIVAMENTE VIM/GEDIT/GEANY. MEM% = RAM USADA NO INSTANTE DE PROVA INDICADO (0 H = INÍCIO; 4 H = ÚLTIMA HORA); SWAP = MÉDIA DAS 5 H DE PROVA. 16 GB·LEVE (N=2) E 32 GB (SWAP ≈ 0) OMITIDOS.

Para enxergar a pressão *durante* a prova, reconstruímos o consumo minuto a minuto de cada máquina, casando-a ao seu time pelo BOCA, e a classificamos pelo **perfil de editor**: o time passou a maior parte da prova (≥60% do tempo) no IntelliJ (IDEA/CLion/PyCharm), no VS Code ou só em editores leves; uso misto fica de fora. No detalhe, o consumo tem forma de U (Fig. 1), alto no início (login e indexação dos servidores de linguagem), recuando até o mínimo por volta de 2 h–2 h30 de prova e subindo de novo nas duas horas finais.

O perfil importa tanto quanto a RAM (Tab. 3). Em 8 GB, times de **IntelliJ paginaram 189 MB** em média, contra 104 do VS Code e 82 dos leves, e já abrem a prova com 46% da RAM ocupada, sinal do JVM carregado cedo. Somar VS Code e IntelliJ num único "pesado" escondia isso, porque o VS Code é muito mais comum (103 contra 27 máquinas) e rebaixa a média. Mais memória absorve a pressão. Em 16 GB a diferença cai (52 contra 43 MB) e em 32 GB o swap é nulo em qualquer perfil. O swap em si é pequeno e não houve travamento [7], mas é só a ponta visível. O sistema só pagina depois de descartar o cache que acelera a compilação, então mesmo 189 MB indicam uma máquina perto do limite. O que de fato aperta é a folga de RAM. Com a prova já começando perto de metade da memória ocupada, sobra pouco para quando o time compila e executa a própria solução. A combinação de IDE pesada e pouca RAM é a que mais pressiona, embora poucos times usem só editores leves (daí o n pequeno desse perfil).

6. Discussão e considerações finais

Do lado técnico, a conta fecha a favor dos 16 GB. A Tab. 3 não deixa dúvida de que máquinas de 8 GB atravessam a prova sob pressão, e nem poderia ser diferente diante do que roda numa máquina de prova: o Ubuntu 26.04 já recomenda 6 GB só para uso confortável [1], as IDEs da JetBrains pedem 8 GB de RAM total [4] e mesmo o VS Code, leve no papel (1 GB [5]), consome vários GB com servidores de linguagem.

Recomendações genéricas de RAM, porém, ajudam pouco aqui. Com a internet bloqueada, o time fica só com o navegador em uma aba (o BOCA) e os editores, somados à própria solução que ele compila e executa para se testar. Os limites do regulamento (1 GB de memória e 128 MB de pilha por solução) são forçados apenas nos testes do juiz; na máquina do time, uma implementação com erro ou uma força bruta usada para conferir respostas pode consumir vários gigabytes. Em 8 GB, é tudo isso disputando a mesma memória.

Por outro lado, 34% dos times da Primeira Fase ainda usam 8 GB, e subir o piso agora, sem plano de transição, tornaria inelegível cerca de um terço deles (11% no Sul, 65% no Norte). Já a queda de 49% para 34% em um único ciclo indica que um cronograma plurianual dá conta do resto. A direção é correta, basta dar tempo às sedes.

Há, porém, um fator conjuntural. Segundo a TrendForce, os preços de contrato da DRAM devem subir entre 58% e 63% só no 2º trimestre de 2026 [8], puxados pela realocação para servidores de IA. Ou seja, o custo de atualizar dispara justamente na hora em que os dados passam a pedir 16 GB.

Para organizadores de sede, vale priorizar RAM sobre CPU, pois a carga de processador é baixa na prova (load de 0,4 a 0,5 com editores pesados e ~0,3 com os leves) e cerca de 3/4 das máquinas já têm CPU de 2017 em diante, enquanto a RAM é o gargalo dominante. **Para a Direção de Sistemas**, o encaminhamento é recomendar desde já que as sedes utilizem máquinas com 16 GB de memória e discutir com a comunidade o cronograma para transformar essa recomendação em exigência.

Referências

[1] Canonical. Ubuntu 26.04 LTS release notes. Disponível em: https://documentation.ubuntu.com/release-notes/26.04/, 2026. Acesso em: maio 2026.

[2] Davi Antônio da Silva Santos, Bruno César Ribas. Maratona Linux a tale of upgrading from Ubuntu 20.04 to 22.04. arXiv preprint arXiv:2510.15263, 2025.

[3] Cassio P. de Campos, Carlos E. Ferreira. BOCA: um sistema de apoio a competições de programação. In: Workshop de Educação em Computação. Sociedade Brasileira de Computação, 2004.

[4] JetBrains. Install IntelliJ IDEA: system requirements. Disponível em: https://www.jetbrains.com/help/idea/installation-guide.html. Acesso em: maio 2026.

[5] Microsoft. Requirements for Visual Studio Code. Disponível em: https://code.visualstudio.com/docs/supporting/requirements. Acesso em: maio 2026.

[6] Wall Berg Morais, Bruno Cesar Ribas. Maratona-Linux: um ambiente para a Maratona de Programação. Anais do Computer on the Beach, p. 416–426, 2019.

[7] The Linux Kernel community. PSI: Pressure Stall Information. Disponível em: https://docs.kernel.org/accounting/psi.html. Documentação do kernel Linux. Acesso em: maio 2026.

[8] TrendForce. AI Server Demand to Drive Memory Contract Price Increases in 2Q26. Disponível em: https://www.trendforce.com/presscenter/news/20260331-12995.html, 2026. Acesso em: maio 2026.



DIAMANTE

Pax

Minha primeira olimpíada de programação foi em 2008 e me moldou até hoje. As olimpíadas não me ensinaram a resposta de nenhum problema em particular, mas sim a coragem de encarar uma questão desafiadora e quebrá-la em partes menores.

Foi justamente com esse espírito que fundamos a Pax. Olhamos para o crime no Brasil – um problema que parecia impossível – e decidimos atacá-lo por partes. Cada nova feature da nossa plataforma é um pedaço desse problema resolvido.

Mas nenhum de nós teria escrito a primeira linha disso sem o que aprendeu em competições como esta. Grande parte do nosso time de engenharia é formada por ex-participantes, finalistas e medalhistas de olimpíadas científicas, muitos deles com passagem pela Maratona SBC e pela ICPC.

Acreditamos que as olimpíadas cultivam o talento capaz de encarar os problemas mais difíceis do país por meio da tecnologia, e como uma empresa brasileira que reconhece e valoriza o talento que se forma aqui, é uma honra apoiar a Maratona SBC.

Phyllipe Medeiros, Chief Technology Officer (CTO) da Pax e ex-participante da Maratona SBC

ELAS TAMBÉM PROGRAMAM, ELAS TAMBÉM VENCEM

A Força da Maratona Feminina de Programação

Imagine entrar em uma competição de programação e, ao olhar ao redor, encontrar dezenas de outras mulheres que, assim como você, estão ali para resolver problemas, enfrentar desafios e provar a si mesmas até onde podem chegar. Mais do que uma disputa contra o relógio, é a oportunidade de perceber que existe um espaço para você na computação, e que você não precisa ocupá-lo sozinha.

É nesse espírito que nasce a Maratona Feminina de Programação (MFP), competição voltada para mulheres e meninas de toda a América Latina, criada com o propósito de contribuir para a redução da desigualdade de gênero na tecnologia e que hoje se encontra na sua 4ª edição. Mais do que reunir participantes diante de problemas algorítmicos, a MFP se propõe a construir uma experiência acolhedora, colaborativa e inspiradora, na qual competição e comunidade caminham lado a lado.

A trajetória da MFP ilustra esse crescimento. Desde a primeira edição, em 2023, quando reuniu 97 participantes, a competição passou a atrair cerca de mil inscritas na primeira fase e 130 finalistas nas edições de 2024 e 2025. Na edição de 2026, a primeira fase, realizada no formato online em 23 de maio de 2026, reuniu participantes de diferentes partes da América Latina e contou com 66 sedes locais, sendo 47 no Brasil e 19 fora dele. Já a segunda fase aconteceu nos dias 1 e 2 de agosto de 2026, no Instituto de Computação da Unicamp, reunindo participantes de diferentes regiões para mais uma etapa de aprendizado, desafios e integração.

Em vez de fazer da distância uma barreira, a competição transformou universidades e instituições em pontos de encontro, fortalecendo comunidades locais e aproximando participantes que talvez nunca tivessem tido a oportunidade de se conhecer.
Na programação competitiva, o desafio costuma ser direto: resolver o maior número possível de problemas no menor tempo. Mas, para muitas participantes da MFP, a experiência não termina quando o cronômetro para. A maratona também cria oportunidades de conhecer estudantes de diferentes países, conversar com professoras e treinadoras, aproximar-se de profissionais e empresas e construir redes que seguem muito além da competição. A própria organização destaca que desses encontros podem nascer amizades, equipes, oportunidades de pesquisa, intercâmbios, estágios e novos caminhos acadêmicos e profissionais.



Edição 2026 Maratona Feminina de Programação

66 SEDES LOCAIS

47 NO BRASIL

17 NO EXTERIOR

Falar sobre a Maratona Feminina de Programação é, também, falar sobre representatividade. Quando uma menina vê outras mulheres programando, competindo e ocupando aquele espaço, a computação deixa de parecer um lugar distante e passa a ser uma possibilidade real para ela. Quando uma estudante encontra uma comunidade que a incentiva, novos caminhos podem se abrir. E quando centenas de mulheres se reúnem para enfrentar os mesmos desafios, a mensagem ultrapassa os limites de uma competição: elas pertencem a esse espaço.

A MFP já impactou centenas de meninas e mulheres e vem construindo redes de apoio que vão muito além da própria competição, talvez o seu maior legado. Afinal, vencer uma maratona pode significar chegar ao primeiro lugar. Mas também pode significar vencer a insegurança, conquistar confiança, fazer uma nova amizade, descobrir uma área de interesse ou simplesmente olhar para uma tela cheia de código e pensar: "eu consigo fazer isso."

Porque elas também programam. Elas também competem. Elas também vencem. E, a cada nova edição, mostram que o futuro da computação também está sendo escrito por elas.



RADIX HEAP: ACELERANDO O DIJKSTRA COM PESOS INTEIROS PEQUENOS

Lucas Pavanelli, FGV EMAp

Resumo

O algoritmo de Dijkstra é onipresente em programação competitiva, e geralmente o implementamos com um binary heap (std::priority_queue do C++). Quando os pesos das arestas são inteiros pequenos, porém, é possível fazer melhor. Este tutorial apresenta o radix heap: uma fila de prioridade que tira proveito do fato de que o Dijkstra extrai as distâncias em ordem não-decrescente. Explicamos como ele funciona, como integrá-lo ao Dijkstra e quando ele compensa.

1. O gargalo do Dijkstra

O problema do caminho mínimo de fonte única é frequente em programação competitiva, e o algoritmo de Dijkstra é a ferramenta padrão quando os pesos são não-negativos. A implementação usual emprega um *std::priority_queue* (um *binary heap*), em que cada inserção e cada remoção do mínimo custam $\mathcal{O}(\log n)$, levando ao total conhecido $\mathcal{O}((m + n) \log n)$.

O Dijkstra, porém, tem uma propriedade que esse *heap* não aproveita. Como o algoritmo extrai os vértices em ordem não-decrescente de distância, as chaves removidas da fila nunca diminuem, e uma fila com essa característica é chamada de monótona. Quando os pesos são inteiros e o maior deles é C, o *radix heap* [1] explora essa monotonicidade para trocar o fator $\log n$ por um fator que depende de $\log C$. Para C pequeno, essa troca compensa.

2. Como funciona o radix heap

A ideia é distribuir as chaves em *buckets* conforme a distância delas até o mínimo corrente. Mantemos o mínimo u (a última chave removida, inicialmente 0) e uma sequência de *buckets*. Uma chave $k \geq u$ vai para o *bucket* de índice $b + 1$, onde b é a posição do bit mais significativo em que k difere de u; o *bucket* 0 fica reservado às chaves iguais a u, daí o deslocamento de um. Por exemplo, com $u = 0$, a chave $13 = 1101_2$ difere no bit de posição 3 e cai no bucket 4 (faixa 8–15), ao lado das chaves 8 e 10. Essas são as distâncias dos vértices a, b e c alcançados a partir da fonte s: a Figura 1 mostra o *radix heap* guardando os nós nesse *bucket*, indexados por suas distâncias.

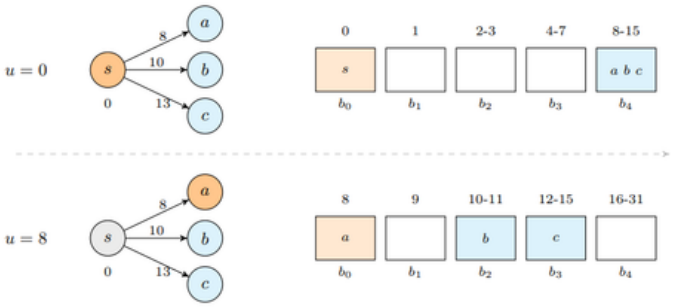


FIGURA 1: DO GRAFO AOS BUCKETS, QUE GUARDAM OS NÓS INDEXADOS PELA DISTÂNCIA. COM u = 0 (ACIMA), A FONTE s OCUPA O BUCKET 0 E SEUS VIZINHOS a, b E c (DISTÂNCIAS 8, 10 E 13) CAEM NO BUCKET 4, DE FAIXA 8–15. AO EXTRAIR a COMO NOVO MÍNIMO (ABAIXO, u = 8), AS FAIXAS PASSAM A VALER A PARTIR DE 8 E OS NÓS b E c MIGRAM PARA BUCKETS MAIS ESTREITOS.

O ponto central é que os buckets têm larguras diferentes, com o *bucket* i cobrindo uma faixa de 2^{i-1} valores acima de u. Os de índice baixo são estreitos; os de índice alto, largos (linha de cima da Figura 1). Como as distâncias nunca passam de nC, bastam $\mathcal{O}(\log(nC))$ *buckets*: 65 cobrem qualquer valor de 64 bits.

O índice resulta de duas operações: um XOR entre a chave e u, seguido de um *count-leading-zeros* (Código 1). O XOR é o truque essencial: ele isola o bit em que k se afasta de u, fazendo a mesma regra valer para qualquer valor de u.

```cpp
std::size_t bucket_index(Weight key) const {
    Weight diff = key - last_deleted_key;
    if (diff == 0) return 0;
    return most_significant_bit(diff) + 1;
}

// most_significant_bit(v) = 63 - __builtin_clzll(v)
```

CÓDIGO 1: ÍNDICE DE BUCKET NO RADIX HEAP.

Inserir uma chave é $\mathcal{O}(1)$, pois basta calcular seu *bucket* e adicioná-la a ele. A extração do mínimo é onde o *bucket* 0 atua. Enquanto ele tem elementos, todos valem u e são removidos diretamente. Quando esvazia, percorremos os buckets em ordem até o primeiro não-vazio, sua menor chave passa a ser o novo mínimo u', e os elementos desse *bucket* são redistribuídos segundo u'. Como esse *bucket* é largo e u' está dentro de sua faixa, as chaves se espalham por *buckets* mais estreitos, enquanto os seguintes ficam intactos. A linha de baixo da Figura 1 mostra esse passo para $u' = 8$.

A eficiência vem de uma invariante simples: uma chave nunca sobe de *bucket*. Cada chave só desce, e o faz no máximo $\mathcal{O}(\log C)$ vezes ao longo de sua vida. Somando sobre as n chaves, o Dijkstra inteiro custa $\mathcal{O}(m + n \log C)$. O $\log n$ do *binary heap* deu lugar a $\log C$.

3. Refinando: dois níveis

Podemos reduzir ainda mais o número de níveis lendo a chave como dígitos numa base $K = 2^b$ (usamos $b = 4$, $K = 16$) em vez de bit a bit. O primeiro nível indica a posição d do dígito que difere do mínimo; o segundo, entre K *sub-buckets*, indica o valor s desse dígito. Retomando os nós a, b e c da Figura 1: com $u = 0$, suas distâncias 8, 10 e 13 são menores que 16, logo têm $d = 0$, e o nível 2 as separa pelos valores $s = 8$, 10 e 13 (Figura 2); uma distância maior, como $300 = 12C_{16}$, ficaria na posição $d = 2$. Assim cada elemento percorre $\mathcal{O}(\log C / \log\log C)$ posições em vez de $\log C$, e o custo cai para $\mathcal{O}(m + n \log C / \log\log C)$. Nos experimentos essa variante supera a de um nível em quase todos os cenários, ao custo de uma implementação mais complexa.

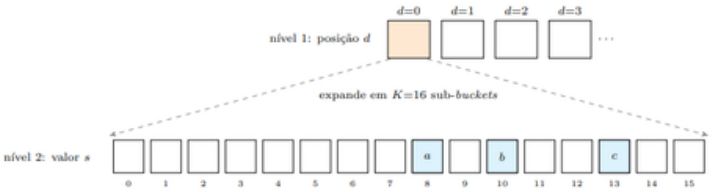


FIGURA 2: RADIX HEAP DE DOIS NÍVEIS (BASE $K = 16$) PARA OS MESMOS NÓS DA FIGURA 1. O NÍVEL 1 AGRUPA AS CHAVES PELA POSIÇÃO d DO DÍGITO QUE DIFERE DE u: COM $u = 0$, AS DISTÂNCIAS 8, 10 E 13 (MENORES QUE 16) TÊM $d = 0$. O NÍVEL 2 ABRE ESSE GRUPO EM K SUB-BUCKETS E SEPARA OS NÓS PELO VALOR s DO DÍGITO, QUE AQUI COINCIDE COM A PRÓPRIA DISTÂNCIA.

4. Como usar

A única exigência do *radix heap* é que as chaves extraídas não decresçam, o que o Dijkstra satisfaz naturalmente, pois a distância d do vértice removido só aumenta. Para adotar o radix heap, basta substituir a *std::priority_queue* por ele e abrir mão do *decrease-key*: em vez de atualizar uma entrada, inserimos uma nova com a chave menor e, ao extrair, descartamos qualquer entrada cuja chave não corresponda mais à distância corrente (*lazy deletion*). O *loop* fica como no Código 2.

```
RadixHeapQueue pq;
dist[s] = 0;
pq.push(0, s);

while (!pq.empty()) {
    auto [d, u] = pq.pop_min();
    if (d != dist[u]) continue;  // entrada obsoleta

    for (auto [v, w] : adj[u]) {
        if (d + w < dist[v]) {
            dist[v] = d + w;
            pq.push(dist[v], v);  // sem decrease-key
        }
    }
}
```

CÓDIGO 2: DIJKSTRA COM RADIX HEAP E LAZY DELETION.

Dois cuidados: os pesos precisam ser inteiros não-negativos, e as distâncias (até nC) devem caber no tipo das chaves (64 bits bastam na prática).

5. Quando vale a pena

O *radix heap* nem sempre é melhor. Medimos o Dijkstra com *binary heap*, *radix* de um nível e *radix* de dois níveis em quatro famílias de grafos, variando C: *grid* (malha 2D esparsa), *sparse* (aleatório com $m \approx 3n$), *dense* (Erdős–Rényi, $m \approx n^2$) e *layered* (DAG em camadas). A Tabela 1 traz o tempo mediano em ms nas colunas *binary*, *radix-1* e *radix-2* (*binary heap* e *radix heaps* de um e dois níveis). Com $C = 10$, um radix vence em todas as famílias, sendo até ~1.7× mais rápido nos grafos esparsos, chegando a ~1.8× com $C = 10^3$. O cruzamento ocorre entre $C = 10^3$ e $C = 10^6$; para pesos grandes, o *std::priority_queue* volta a vencer.

Família	C	binary	radix-1	radix-2
dense (n=2k)	10	1.03	**0.93**	0.96
	1 000	1.30	1.24	**1.08**
	1 000 000	**1.35**	1.98	1.73
grid (n=360k)	10	17.52	14.73	**13.28**
	1 000	21.31	41.48	**21.22**
	1 000 000	**21.43**	59.59	51.41
layered (n=120k)	10	4.41	2.99	**2.82**
	1 000	6.02	8.26	**4.74**
	1 000 000	**5.66**	13.67	12.54
sparse (n=250k)	10	19.47	**11.47**	11.76
	1 000	25.35	15.19	**13.89**
	1 000 000	**26.03**	44.13	31.29

TABELA 1: TEMPO MEDIANO DE EXECUÇÃO (MS) POR FILA DE PRIORIDADE. MELHOR VALOR POR LINHA EM NEGRITO.

Como regra prática, vale trocar pelo *radix* de dois níveis quando os pesos são pequenos ($C \lesssim 10^3$) e o grafo é grande e esparso. Para pesos grandes ($C \geq 10^6$) ou grafos pequenos e densos, prefira o *binary heap*, mais simples e mais rápido.

6. Considerações finais

O *radix heap* é uma substituição direta da *std::priority_queue* que pode acelerar consideravelmente o Dijkstra quando os pesos são inteiros pequenos, cenário comum na competição. A ideia de organizar chaves em *buckets* remonta ao algoritmo de Dial [3], que o *radix heap* refina [1]; o levantamento de Costa et al. [2] cobre estas e outras filas monótonas. A implementação completa em C++, com os geradores de grafos e os experimentos, está em https://github.com/pavalucas/monotone-dijkstra.

Referências

[1] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. Journal of the ACM, 37(2):213–223, 1990.

[2] V. Costa, M. Castro, and R. de Freitas. Exploring monotone priority queues for dijkstra optimization. RAIRO Operations Research, 59, 2025.

[3] Robert B. Dial. Algorithm 360: Shortest-path forest with topological ordering. Communications of the ACM, 12(11):632–633, 1969.

TEORIA DE ROLES PARA FORMAR EQUIPES DE ICPC

Racsó Galván, UNICAMP
Miguel Miní, UNICAMP

Resumo

Este artigo propõe uma teoria prática de roles para formar equipes orientadas ao ICPC. A ideia central é que selecionar uma equipe apenas com base no nível individual ou na afinidade entre seus integrantes nem sempre leva ao melhor desempenho em um contest com um único computador, no qual a distribuição de responsabilidades exerce papel fundamental. A partir dos atributos de força e velocidade, organizamos uma visão simplificada do contest em três etapas e descrevemos os roles de Mid, Carry e Tank, apresentando recomendações para a formação e o treinamento de equipes.

1. Introdução

A formação de equipes para competições de programação costuma priorizar o nível individual dos competidores ou a afinidade entre eles. Entretanto, nossa experiência como competidores e treinadores sugere que esses critérios, por si só, nem sempre são suficientes para obter o melhor desempenho coletivo. Em um contest do ICPC, no qual três integrantes compartilham um único computador, a forma como as responsabilidades são distribuídas durante a prova pode ser tão importante quanto a habilidade individual de cada membro.

A ideia apresentada neste trabalho surgiu da observação de equipes ao longo de anos de treinamento e competição. Inspirados na noção de papéis especializados presente em esportes coletivos e jogos em equipe, propomos uma teoria prática para organizar equipes de ICPC em funções complementares. O objetivo é oferecer um referencial simples para auxiliar na formação, no treinamento e na análise de equipes. A proposta tem caráter prático e não busca constituir um modelo matemático de otimização.

2. Atributos

Para estabelecer uma visão simplificada do conjunto de habilidades de um competidor, definiremos força e velocidade como os atributos principais para a escolha de roles em uma equipe ideal.

2.1. Força

A força de um competidor refere-se à capacidade de resolver problemas que exigem uma análise mais profunda. Em geral, essa característica se manifesta em problemas de estilo olímpico ou nos problemas de maior dificuldade em uma competição de ICPC.

Indicadores numéricos possíveis são:
- Dificuldade média (em *rating*) dos problemas resolvidos.
- Quantidade média de problemas resolvidos em competições de alta dificuldade (Codeforces Div. 1, AtCoder AGC etc.).

Um indicador central para identificar um competidor com alta força é sua capacidade de dedicar um longo período de tempo à análise de um mesmo problema, explorando múltiplas camadas de raciocínio até encontrar uma solução. Na prática, essa avaliação pode combinar o histórico de problemas resolvidos com a observação de simulados: registram-se as pessoas que formulam ideias, invariantes ou reduções para os problemas mais difíceis, inclusive quando a implementação final fica a cargo de outro integrante. Esses indicadores são orientadores e não constituem uma medida universal de força.

2.2. Velocidade

A velocidade de um competidor refere-se à capacidade de realizar rapidamente ações relevantes para a resolução de problemas.

Indicadores numéricos possíveis são:
- Tempo médio necessário para ler enunciados.
- Alta velocidade de digitação (*words per minute* — WPM).
- Tempo médio necessário para analisar um problema de um nível compatível com o do competidor.

O principal critério para identificar um competidor com alta velocidade é sua capacidade de obter uma vantagem significativa de tempo e penalidade em relação a competidores de nível semelhante. Em simulados, esse atributo pode ser acompanhado pelo tempo até o primeiro problema aceito, pelo tempo entre uma ideia validada e a submissão aceita e pela quantidade de problemas acessíveis corretamente resolvidos nos minutos iniciais. Também aqui, os valores devem ser interpretados como evidências informais, e não como uma classificação rígida.

Os dois atributos são complementares. A estratégia de seleção pode dar maior peso a um deles, desde que as funções críticas do contest estejam cobertas. O balanceamento perfeito dos atributos não é um requisito.

3. Estruturando o contest

Para introduzir os roles principais, é útil separar os momentos de uma competição de acordo com a dificuldade média dos problemas resolvidos.

3.1. Etapas de uma competição

De forma semelhante ao que acontece em alguns jogos em equipe, definiremos as três fases a seguir:
1. *Early game*: etapa inicial, dominada por problemas fáceis e pela necessidade de acumular a menor penalidade possível.
2. *Mid game*: etapa de equilíbrio, na qual predominam problemas intermediários e o mais importante é manter ou recuperar a posição relativa.

3. *Late game*: etapa de definição, na qual restam os problemas mais duros e, em geral, ocorre a hora congelada, com o foco em consolidar a vantagem por meio de soluções.

Essa divisão é uma aproximação que ajuda a entender por que certos perfis pesam mais em certos momentos.

4. Roles

Uma equipe pode se apoiar mais em um atributo do que no outro; ainda assim, vale começar pela situação ideal.

4.1. Roles principais

Os roles principais seguem a estrutura do contest descrita acima:
• *Mid*: responsável pelo *early game*. Seu principal atributo deve ser a velocidade, e espera-se que apresente a maior velocidade e a menor força relativa entre os três integrantes.
• *Carry*: responsável pelo *mid game*. Seus atributos devem ser mais equilibrados, e espera-se que apresente força superior à do Mid e velocidade superior à do Tank.
• *Tank*: responsável pelo *late game*. Seu principal atributo deve ser a força, e espera-se que apresente a maior força e a menor velocidade relativa entre os três integrantes.

A associação é direta: o Mid limpa o que é acessível com precisão, o Carry fecha problemas médios sem perder ritmo e faz a transição natural do contest, e o Tank sustenta a parte mais difícil da prova. Nesse sentido, o Carry converte ideias em Accepted e conecta o *early game* ao *late game*.

Em uma equipe ideal com uma pessoa em cada role principal, a distribuição por etapa tende a ser a seguinte:
• *Early game*: *Mid* + *Carry* + *Tank*, isto é, cada um resolve por conta própria, a menos que algum problema justifique análise conjunta.
• *Mid game*: (*Mid*, *Carry*) + *Tank*, isto é, o Mid se junta ao Carry para atacar os problemas intermediários, enquanto o Tank segue por conta própria, salvo quando o problema pede o esforço dos três.
• *Late game*: (*Mid*, *Carry*, *Tank*), isto é, o Tank lidera a resolução com apoio do Mid e do Carry. Com o início da hora congelada, o agrupamento (*Mid*, *Carry*, *Tank*) tende a ser o mais recomendado.

A distribuição admite que qualquer integrante resolva problemas fora de sua etapa. A dinâmica da equipe depende de comunicação e de troca de tarefas.

4.2. Configuração da equipe

A teoria de roles orienta a configuração de equipes. Uma configuração saudável precisa responder a três perguntas. Quem limpa o set e evita penalidade barata? Quem transforma ideias razoáveis em solução aceita? Quem sustenta os problemas que exigem mais profundidade?

Se dois ou três jogadores querem ocupar o mesmo espaço dentro do contest, podem surgir desordem, leitura duplicada, problemas sem dono claro ou implementações tardias. Por isso, até uma configuração com três jogadores de altíssimo nível pode falhar se as funções não estiverem bem distribuídas.

Alguns padrões de falha aparecem com frequência. Três jogadores muito parecidos podem duplicar leitura e desperdiçar tempo. Dois perfis muito profundos sem alguém que limpe o set podem deixar pontos baratos na mesa. Dois jogadores rápidos sem um bom fechador podem gerar boas ideias, mas converter pouco. Em geral, quando falta complementaridade, a equipe joga abaixo do seu nível individual.

4.3. Roles alternativos

Todos os roles alternativos derivam de um papel de apoio.
• *Support*: role de apoio associado a pessoas que ampliam o trabalho coletivo.
• *Tester*: responsável por criar casos de teste feitos à mão para validar soluções antes da submissão.
• *Scribe*: responsável por manter códigos do *Team Reference Document* e algoritmos clássicos.
• *Search engine*: responsável por conhecer muitos algoritmos ou estruturas menos comuns.
• *Light carry/tank*: uma variação de Carry ou Tank com velocidade acima do esperado, à custa de um pouco menos de força.
• *Hard mid/carry*: uma variação de Mid ou Carry com força acima do esperado, à custa de um pouco menos de velocidade.

5. Um experimento de formação

Uma forma de avaliar esta teoria é observá-la na prática. Em nossa própria experiência de treinamento, organizamos a base da equipe *UnratedLegendaryGrandMasters* com Emanuel Soto, Mijail Poccohuanca e Joel Chávez, cujos nicks no Codeforces são cegax, mika_uwu e Joel_Uwu. A definição deliberada das funções guiou a formação da equipe.

Nessa configuração, Soto ocupava o papel de *Hard Mid*, responsável por dar velocidade ao início da prova, limpar o set e evitar desperdício de penalidade em problemas mais acessíveis. Poccohuanca concentrava o perfil de *Tank*, assumindo os problemas de maior profundidade e a parte mais pesada do contest. Chávez cumpria a função de *Carry*, ligando o início da prova aos problemas intermediários, sustentando a transição entre etapas e mantendo o ritmo da equipe quando o contest deixava de ser apenas limpeza e passava a exigir mais conversão. A equipe, portanto, foi montada com uma divisão funcional clara entre limpeza, transição e profundidade.

A trajetória ajuda a dimensionar esse experimento. Em 2023, uma base semelhante ainda aparecia em torno da 30ª colocação no Regional Latino-Americano 2023. A configuração de papéis incluía Ricardo Ulloa como *Support* [4]. Na Programadores de América de 2024, o grupo ficou em 11° lugar [6]. No fim desse ano, já com a definição de roles mais bem consolidada e coerente, a equipe alcançou o 12º lugar no Regional Latino-Americano de 2024 [5] e, depois, o 3º lugar na Programadores de América de 2025 [7].

Este estudo de caso apresenta resultados consistentes com a hipótese de que uma distribuição clara de funções pode contribuir para o desempenho em ICPC. A distribuição explícita de responsabilidades é um elemento central da proposta.

5.1. Trajetória da equipe UNI-FC

Um caso histórico que motivou esta discussão é o da equipe UNI-FC, formada por Miguel Miní (Bashca), Víctor Racsó (racsosabe) e Sergio Sánchez (Valgt). Como *Gracias*, *Marco*, o trio ficou em 32º lugar no placar global do Regional Latino-Americano de 2018, resolvendo sete problemas [1]. No Regional Latino-Americano de 2019, já como *Rating MiSeRable*, alcançou o 14º lugar global, também com sete problemas resolvidos [3]. Posteriormente, a equipe obteve 25º lugar, com quatro problemas resolvidos entre treze, no ICPC World Finals Invitational 2020, realizado on-line em 2021, com a distinção *Honors* [2].

Segundo o relato dos autores, o trio foi formado aproximadamente duas semanas antes de sua primeira classificação regional por meio de um processo em quatro etapas. Primeiro, realizou-se um concurso individual de força, usando a quantidade de problemas resolvidos em diferentes níveis como indicador inicial. Em seguida, os candidatos competiram em duplas em contests de nível regional. A partir das melhores duplas, o terceiro integrante foi escolhido por uma busca exaustiva das formações possíveis; quando havia mais de um candidato viável ao terceiro posto, um confronto adicional de força foi usado como desempate. O procedimento reuniu evidências de desempenho individual, desempenho coletivo e complementaridade entre os candidatos.

O processo oferece uma maneira concreta de selecionar por habilidades e avalia diretamente força e desempenho em dupla.

Os resultados documentam a trajetória da equipe, sem isolar a contribuição de cada elemento do processo de formação. Esta seção apresenta uma evidência qualitativa compatível com a hipótese de complementaridade de funções. Treinamento, experiência acumulada, qualidade dos adversários e variação entre provas compõem o contexto desses resultados.

6. Limitações

Os resultados apresentados neste trabalho baseiam-se em um único estudo de caso, correspondente à equipe dos próprios autores. O alcance das conclusões está restrito a esse contexto. Nível técnico, experiência prévia, estilo de comunicação e dinâmica entre os integrantes influenciam o desempenho.

O artigo apresenta a teoria de roles e o estudo de caso descrito. Uma comparação sistemática entre equipes requer dados e critérios homogêneos, que poderão ser reunidos em estudos futuros.

Estudos futuros poderão ampliar essa validação por meio da análise sistemática de um número maior de equipes e competições.

7. Conclusões

Este trabalho propôs uma teoria prática de roles para a formação de equipes de ICPC, baseada na ideia de que diferentes perfis de competidores podem exercer funções complementares durante um contest. A proposta enfatiza a importância de analisar como as características de cada integrante interagem ao longo da competição.

O estudo de caso apresenta uma associação entre a distribuição deliberada de funções e o desempenho coletivo. A relação causal requer validação em um conjunto maior de equipes. Essa forma de análise oferece uma perspectiva útil para treinadores e competidores nos processos de formação, treinamento e identificação de pontos fortes e pontos a desenvolver.

Esperamos que esta teoria incentive novas investigações sobre a dinâmica de equipes em programação competitiva e que estudos futuros ampliem sua validação por meio da análise de diferentes equipes, competições e contextos.

Referências

[1] ICPC Latin America. ICPC Latin America Regional Contest 2018 — Final Scoreboard. Disponível em: https://www.ime.usp.br/~maratona/images/brasileira-2018/placar.pdf. Acesso em 30 jul. 2026.
[2] ICPC World Finals Invitational. ICPC WF Moscow Invitational Contest — Final Standings. Disponível em: https://wfi.icpc.global/wp-content/uploads/2021/10/Codeforces1.pdf. Acesso em 30 jul. 2026.
[3] ScoreLATAM. ICPC Latin America Regional Contest 2019 — Final Scoreboard. Disponível em: https://scorelatam.naquadah.com.br/latam-2019/. Acesso em 30 jul. 2026.
[4] ScoreLATAM. ICPC Latin America Regional Contest 2023 — Scoreboard. Disponível em: https://scorelatam.naquadah.com.br/latam-2023/. Acesso em 23 jul. 2026.
[5] ScoreLATAM. ICPC Latin America Regional Contest 2024 — Scoreboard. Disponível em: https://scorelatam.naquadah.com.br/latam-2024/. Acesso em 23 jul. 2026.
[6] ScoreLATAM. Programadores de América 2024 — Scoreboard. Disponível em: https://scorelatam.naquadah.com.br/pda24/. Acesso em 23 jul. 2026.
[7] ScoreLATAM. Programadores de América 2025 — Scoreboard. Disponível em: https://scorelatam.naquadah.com.br/pda25/. Acesso em 23 jul. 2026.



DIAMANTE

rivio

O que continua funcionando depois

Toda empresa que patrocina esta revista diz que está em busca de talento. Eu queria usar esse espaço para uma pergunta um pouco diferente: com que critério o talento brasileiro escolhe os desafios que vai encarar, e onde ele acaba parando?

Nesta edição há um estudo sobre os medalhistas de ouro entre 1996 e 2020. Somados, 57,5% foram para big techs e para a indústria de software, e 20,8% estão hoje nos Estados Unidos ou no Canadá. São escolhas ótimas, e não tenho nenhuma crítica a fazer a elas. Mas o gráfico deixa uma pergunta no ar: quais sistemas brasileiros precisavam desse nível de gente e nunca tiveram?

A saúde é um deles. O país gasta cerca de 10% do PIB em saúde e perde por ano algo próximo de 28 bilhões de dólares em ineficiência hospitalar. É muito dinheiro parado num lugar onde ele faria diferença imediata.

Quando se fala em saúde, a gente pensa logo na enfermagem, nos médicos, na diferença que eles fazem. E fazem mesmo, não tenho dúvida. Só que boa parte do que trava um hospital hoje acontece longe do leito, no intervalo entre o fim do tratamento e o dia em que o pagamento chega. Esse intervalo é um problema de engenharia.

Hoje o setor financeiro de um hospital leva quarenta horas de trabalho humano para auditar cem contas. São quase oitenta dias entre o atendimento ao paciente e o dinheiro entrar. Tem hospital atrasando a folha porque a receita que ele já ganhou está presa numa disputa com o plano de saúde. Nenhum desses gargalos se resolve com mais medicina. Eles se resolvem com casamento de registros, extração de informação de texto sem estrutura, motor de regras e decisão automatizada em escala. É engenharia difícil, do tipo que ainda ninguém resolveu de verdade, nem aqui nem lá fora.

Legado é o que continua funcionando depois que você sai da sala. A medalha que vocês levarem daqui é de vocês para sempre, e ninguém tira. Mas o que vai ficar da geração de vocês é o sistema que vocês deixarem rodando. Hospitais eficientes salvam mais vidas, e hoje boa parte dessa eficiência se escreve em código. Escolham bem onde vão apontar o que vocês sabem fazer. Se um dia quiserem apontar para dentro de um hospital, a porta aqui está aberta.

Ricardo Sales é co-fundador da Rivio, que transforma a gestão de hospitais brasileiros com inteligência artificial.

1ª FASE EM NÚMEROS



965 TIMES

2895 ESTUDANTES COMPETIDORES

233 INSTITUIÇÕES

52 SEDES SIMULTÂNEAS



MENINAS SÃO 18,51%

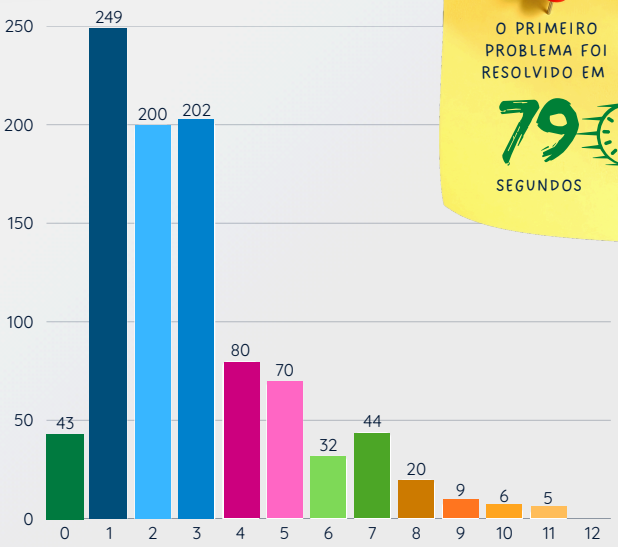
80 TIMES FEMININOS
56 TIMES COM 2 MENINAS
184 TIMES COM 1 MENINA

MAIOR SEDE: SÃO PAULO (SP) - 50 TIMES
MENOR SEDE: BOA VISTA (RR) - 1 TIME

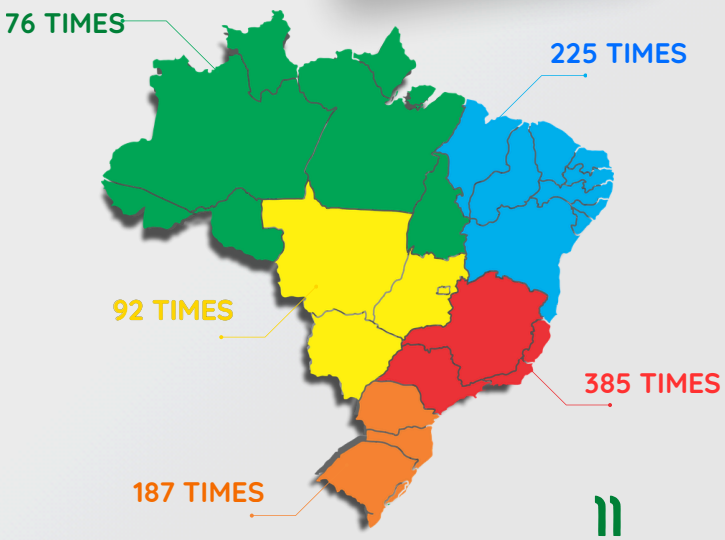




Times por números de problemas



O PRIMEIRO PROBLEMA FOI RESOLVIDO EM **79** SEGUNDOS



CLASSIFICAÇÃO

RANK	BANDEIRA	TIME	TOTAL	PEN.
1		[UFU] Praise the Sonic	11	898
2		[UNICAMP] Enemy Leo used Appeal! It's super effective! Pikachu fainted!	11	1014
3		[UF Santa Catarina] 3 macacos infinitos com heurística	11	1049
4		[UFPE] Shadow, o ouriço-cacheiro-viajante	11	1239
5		[UNICAMP] Los Pitufos	11	1352

11 O MAIOR NÚMERO PROBLEMAS

O PROBLEMA "CONTINGÊNCIA" FOI TENTADO **2990** VEZES **WA**

EM MÉDIA **2,9** PROBLEMAS RESOLVIDOS POR TIME

2781 SUBMISSÕES ACEITAS

UM PROBLEMA INTERESSANTE SOBRE MEDIANAS

Ivan Henrique Costa Petrin, UNICAMP

Resumo

Comento sobre o problema Crystal Peak de um contest longo (10 dias) do grupo GoForGold no codeforces. Esse problema tem um resultado muito interessante sobre medianas e, segundo os autores, era o segundo problema mais difícil da prova. A solução esperada é baseada em um uso incomum de busca binária.

1. Contexto do contest

Na Índia, existe uma iniciativa chamada GoForGold, que pretende apoiar times indianos a conquistarem uma medalha de ouro no ICPC. A primeira vez em que ouvi falar dessa iniciativa foi no portal do codechef, há vários anos. Apesar das empresas que encabeçam a iniciativa terem mudado, o nome continuou o mesmo e expandiram a ideia para o time indiano na IOI.

Essa iniciativa organiza semanas de treinamento, similares às escolas de verão e inverno que temos no Brasil, e também oferece uma premiação em dinheiro caso algum time ou indivíduo consiga trazer a medalha de ouro para o solo indiano.

Em janeiro de 2025, convidaram os usuários do codeforces a participarem de contests longos mensais [2], cada contest com duração de 10 dias. Li os problemas de janeiro e, apesar de não serem necessariamente originais, todos os enunciados foram reformulados para remeterem a alguma parte do universo do jogo Hollow Knight [3]. Como esse é um dos meus jogos preferidos, resolvi tentar resolver os problemas e não me decepcionei porque encontrei um muito bonito e memorável com o título Crystal Peak.

2. Enunciado

Existe uma colina de cristal com N níveis. O nível i contém $2i - 1$ números. Também é dado um array a com $2n - 1$ números, esse é o nível mais baixo da colina. Como exemplo, essa é uma estrutura com $N = 4$ e $a = [5, 1, 4, 7, 6, 2, 3]$.

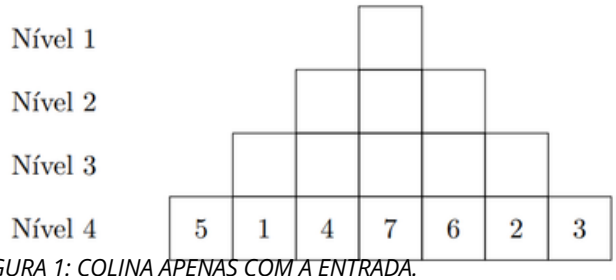


FIGURA 1: COLINA APENAS COM A ENTRADA.

O valor de cada célula nos níveis 1 a $N - 1$ são definidos como a mediana das três células abaixo. No exemplo anterior, essa é a colina preenchida:

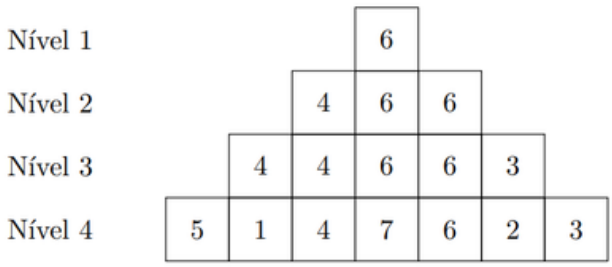


FIGURA 2: COLINA COM TODAS CÉLULAS PREENCHIDAS E PICO = 6.

O elemento no nível 1 é o topo da colina. Seu objetivo é calcular qual o topo da colina. Para o exemplo acima, seu código deve imprimir 6.

Restrições:
• $1 \le N \le 10^5$.
• $-10^9 \le a_i \le 10^9$, os elementos do array a, apenas números inteiros.

3. Solução ingênua

A mediana de um array com uma quantidade ímpar de elementos, é o elemento do meio do array quando ele está ordenado:

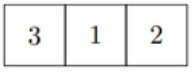


FIGURA 3: O ARRAY ORDENADO É [1, 2, 3]. ASSIM, SUA MEDIANA É 2.

Uma primeira ideia é preencher todas as células, desde o sopé até o topo para encontrar a resposta. Ordernar 3 números para encontrar a mediana leva tempo $O(1)$ mas devemos calcular muitos valores para chegar até o topo.

A colina com N níveis tem

$1 + 3 + 5 + ... + 2n - 1 = \sum_{k=1}^{N} (2k - 1) = N^2$ células.

Assim, mesmo se desconsiderarmos o último nível com $2n - 1$ elementos (que já vem preenchido na entrada), ainda precisaríamos calcular $O(N^2)$ células. Como $N \le 10^5$, essa solução não passa no tempo.

4. Um problema mais fácil

A verdade, caro leitor, é que não fui muito honesto em relação ao enunciado. As restrições acima são para a versão difícil do problema, porém, existia uma versão mais fácil com restrições menores:

Restrições, versão fácil:
• $1 \le N \le 10^5$.
• $0 \le a_i \le 1$, os elementos do array a, apenas números inteiros.

Essas restrições não ajudam a melhorar a nossa solução anterior porque ela só dependia de N e não de a_i. No entanto, com a_i assumindo poucos valores possíves, podemos testar alguns casos manualmente e procurar por alguma propriedade que nos ajude.

Uma observação interessante é que, se existem 2 números iguais lado a lado, esses números flutuam, em suas colunas, do nível mais baixo até o mais alto, independentemente dos elementos adjacentes a eles:

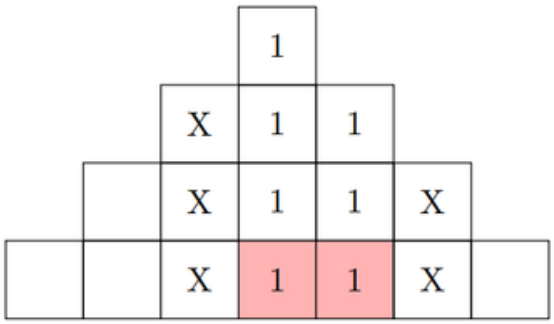


FIGURA 4: X PODE ASSUMIR QUALQUER VALOR, E O NÚMERO 1 É EMPURRADO PARA CIMA NAS 2 COLUNAS CENTRAIS. PODEMOS IGNORAR OS OUTROS NÚMEROS PARA CALCULAR QUALQUER CÉLULA NESSAS DUAS COLUNAS.

Mas o que acontece quando estamos na borda da colina e o vetor do nível acima diminui de tamanho? Primeiro, vamos assumir que não existem dois valores iguais mais próximos da coluna central (a coluna do pico da colina, i.e., da coluna com nossa resposta), porque dois números iguais, lado a lado, bloqueiam aquela coluna com aqueles valores:

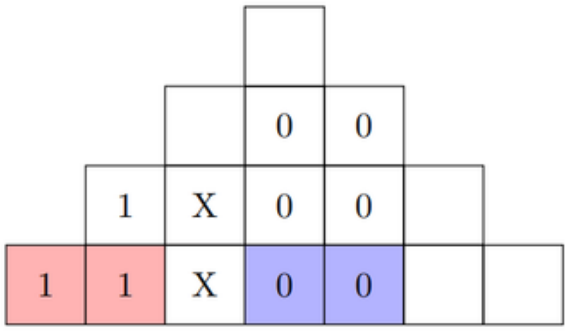


FIGURA 5: OS NÚMEROS 1 NÃO CONSEGUEM INFLUENCIAR A COLUNA CENTRAL PORQUE OS DOIS 0 BLOQUEIAM AQUELA COLUNA, MESMO QUANDO $X = 1$.

Nesse caso, os outros números formam uma sequência alternada entre 0s e 1s e os valores iguais, mesmo na borda da colina, deslizam para perto da coluna central. Veja como os dois números 1 deslizam para a coluna central quando a = [1, 1, 0, 1, 0, 1, 0]:

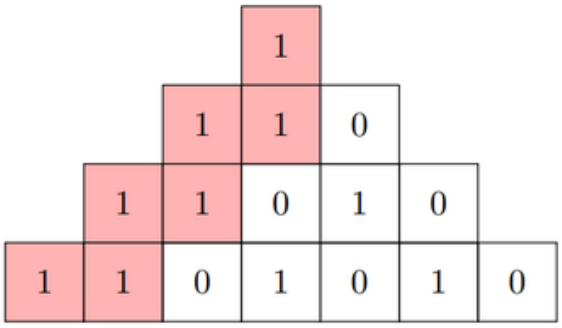


FIGURA 6: OS DOIS NÚMEROS 1 À ESQUERDA SÃO EMPURRADOS PARA CIMA MAS AGORA DESLIZAM RUMO AO CENTRO.

Por essas observações, nossa resposta será o número mais próximo da coluna central desde que ele seja adjacente a algum valor igual a ele.

Para termos um algoritmo, precisamos apenas resolver o caso em que a sequência de entrada é alternada. Nesse caso, precisamos saber apenas a quantidade de níveis na nossa colina. Se a quantidade de níveis é par, temos o complemento do elemento no centro do array a, se a quantidade de níveis é ímpar, é o próprio elemento:

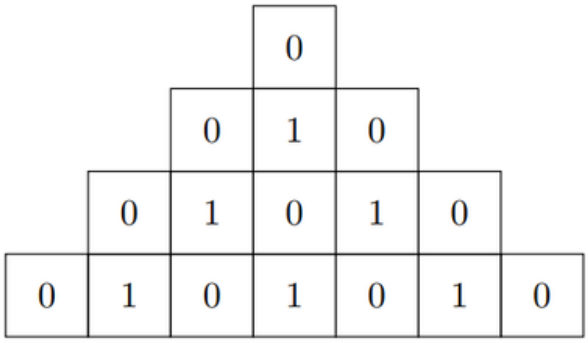


FIGURA 7: A SEQUÊNCIA DE ENTRADA CONTINUA ALTERNADA QUANDO SOBE OS NÍVEIS DA COLINA.

Após toda essa análise, conseguimos resolver a versão fácil do problema, com $O(N)$ tempo. Por praticidade, usamos a 1-indexado, i.e., a_{index} ∈ [1, $2n$ − 1].

```cpp
int solve_easy(vector<int>& a, int n) {
    int leftside_double_idx = -INF;

    for (int i = n; i >= 1; i--) {
        if (a[i] == a[i - 1]) {
            leftside_double_idx = i;
            break;
        }
    }

    int rightside_double_idx = INF;

    for (int i = n; i < 2 * n; i++) {
        if (a[i] == a[i + 1]) {
            rightside_double_idx = i;
            break;
        }
    }

    int dist_left_double = n - leftside_double_idx;
    int dist_right_double = rightside_double_idx - n;

    int min_dist = INF;
    int result_idx = -1;

    if (dist_left_double <= dist_right_double) {
        min_dist = dist_left_double;
        result_idx = leftside_double_idx;
    } else {
        min_dist = dist_right_double;
        result_idx = rightside_double_idx;
    }

    if (min_dist <= n) {
        return a[result_idx];
    } else {
        return get_alternating_answer(a, n);
    }
}

int get_alternating_answer(vector<int>& a, int n) {
    if ((n - 1) / 2 % 2 == 0)
        return a[n];
    else
        return 1 - a[n];
}
```

CÓDIGO 1: SOLUÇÃO DA VERSÃO FÁCIL.

5. Resolvendo a versão difícil

Uma propriedade da mediana é que não importa quais os elementos do array, apenas a ordem relativa entre eles. Isso é muito útil para nós e é o truque que nos permite resolver o problema.

Se propomos que o resultado é ≤ *guess*, podemos transformar o array a de entrada de modo a conter apenas 0 ou 1. Os valores $a_i \leq guess$ se transformam em 0 e os valores $a_i > guess$ se transformam em 1. Calculamos a resposta usando nossa solução da versão fácil, adaptando nosso chute de acordo com o valor no pico da colina. Qual o melhor valor para um chute? O meio do array é uma boa alternativa porque podemos sempre descartar metade do array ao fazer um chute e uma verificação:

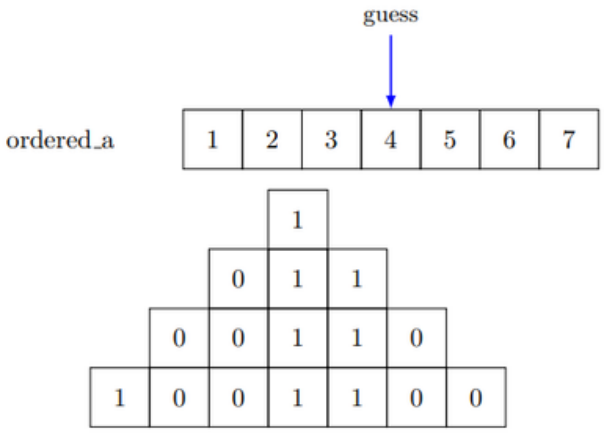


FIGURA 8: TRANSFORMAÇÃO DA COLINA DA FIGURA 1 QUANDO guess = 4 E SEU PREENCHIMENTO. COMO O RESULTADO É 1, A RESPOSTA É MAIOR QUE 4.

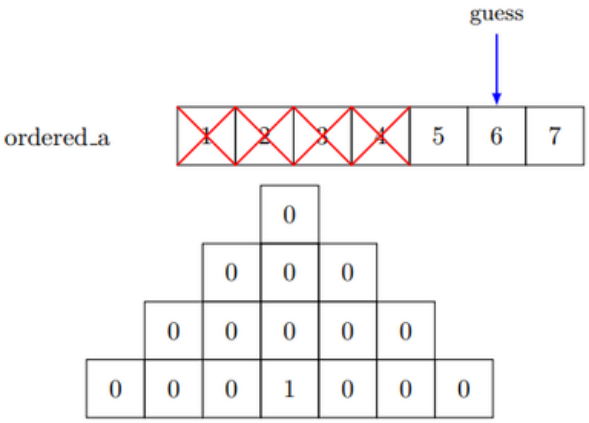


FIGURA 9: TRANSFORMAÇÃO DA COLINA DA FIGURA 1 QUANDO *guess* = 6 E SEU PREENCHIMENTO. COMO O RESULTADO É 0, A RESPOSTA É 6 (*guess*) OU MENOS.

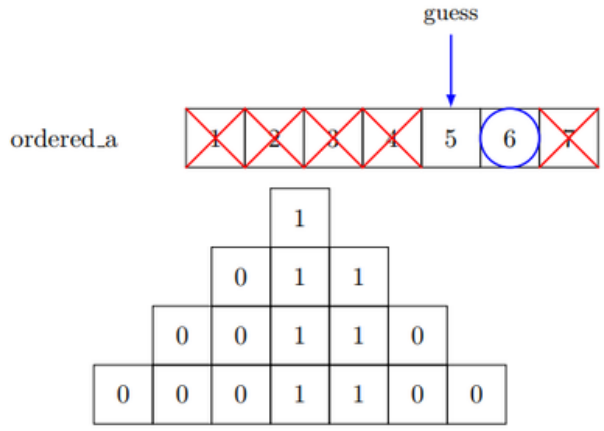


FIGURA 10: TRANSFORMAÇÃO DA COLINA DA FIGURA 1 QUANDO *guess* = 5 E SEU PREENCHIMENTO. COMO O RESULTADO É 1, A RESPOSTA É MAIOR QUE 5.

```cpp
int solve_hard(vector<int>& a, int n) {
    vector<int> ordered_a = get_ordered(a);

    int low = 1;
    int high = 2 * n - 1;
    int best = 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        vector<int> simpler_problem_a =
            build_simpler_array(a, n, ordered_a[mid]);
        int cur_result = solve_easy(simpler_problem_a, n);

        if (cur_result == 0) {
            best = mid;
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }

    return ordered_a[best];
}

vector<int> build_simpler_array(vector<int>& a, int n, int val
    vector<int> simpler_problem_array(2 * n);

    for (int i = 1; i < 2 * n; i++) {
        if (a[i] <= val) {
            simpler_problem_array[i] = 0;
        } else {
            simpler_problem_array[i] = 1;
        }
    }

    return simpler_problem_array;
}
```

CÓDIGO 2: SOLUÇÃO DA VERSÃO DIFÍCIL.

A complexidade total fica a complexidade da ordenação mais a busca binária com $solve_easy$ dentro dela. Como $solve_easy$ é $O(N)$, a complexidade total de tempo é $O(N \log N)$.

6. Implementação

A implementação completa encontra-se em https://pastebin.com/idaSMmjS.

7. Considerações finais

O que torna esse problema interessante para mim é a simplicidade do enunciado e o processo de transformação da versão difícil para a versão fácil.

Ainda, com exceção das ideias e exploração inicial, para resolver esse problema precisávamos apenas de conhecimento sobre busca binária, que costuma ser um dos primeiros algoritmos ensinados em cursos de computação. Isso, para mim, aumenta ainda mais a beleza do problema.

Por fim, em 2026, foi publicado um blog no codeforces com uma breve descrição dessa ideia e problemas relacionados [1].

Referências

[1] Arvind "arvindr9" Ramaswami. A technique for some median-related problems. Disponível em: https://codeforces.com/blog/entry/154631. Acesso em: 21 jun. 2026.
[2] Naveen "evenvalue" Kulkarni. Invitation to goforgold long challenge jan 2025. Disponível em: https://codeforces.com/blog/entry/138266. Acesso em: 21 jun. 2026.
[3] Hollow knight. Disponível em: https://en.wikipedia.org/wiki/Hollow_Knight. Acesso em: 21 jun. 2026.

UNB NA FINAL NACIONAL DA MARATONA SBC: 2020-2025

Bruno C. Ribas, UnB
Daniel de P. Porto, UnB
Daniel S. N. Nunes, IFB
Edson A. da C. Júnior, UnB
Guilherme N. Ramos, UnB
Vinicius R. P. Borges, UnB

Resumo

Neste artigo analisamos as edições de 2020 a 2025 da Final Brasileira da Maratona SBC de Programação para traçar o perfil competitivo dos times da Universidade de Brasília (UnB). Os 77 problemas foram analisados por tema, dificuldade e nível da técnica exigida para a solução, e associados ao desempenho da instituição. A UnB resolveu 82% dos problemas curriculares, mas apenas 4% dos extracurriculares, o que indica uma possível lacuna em relação aos melhores times brasileiros e pode nortear a preparação para um melhor desempenho futuro.

1. Introdução

A Maratona SBC de Programação é a etapa nacional do *ICPC Latin America*. Os times brasileiros competem sobre o mesmo conjunto de problemas dos demais países da região, e a classificação restrita ao Brasil define a Final Nacional. Considerando os seis eventos de 2020 a 2025, a Universidade de Brasília (UnB) participou com times de dois *campi* distintos, Darcy Ribeiro (identificado no placar por [UnB]) e Faculdade de Ciências e Tecnologias em Engenharias (FCTE, antiga [UnB/FGA]), que participam de forma independente. Apesar do desempenho de alto nível nos últimos anos, faltava um diagnóstico objetivo para direcionar as ações da instituição: o que caracteriza os problemas que os times resolvem e os que não? Estas perguntas são respondidas neste trabalho com base nos placares oficiais[1], cruzando o desempenho de cada time da UnB com uma categorização dos problemas por tema, dificuldade e nível da técnica necessária para a solução[2].

2. Dados e Metodologia

Os placares foram coletados e, para cada ano e cada time, extraiu-se os problemas *resolvidos*, os *tentados sem sucesso* e os *não tentados*, além da colocação final e penalidades. Times convidados foram excluídos do ranking oficial brasileiro.

Cada um dos 77 problemas foi rotulado pelos autores, a partir dos enunciados e dos *editoriais* oficiais, segundo três eixos. O primeiro é o tema, composto por 11 categorias, como *ad-hoc* e *estruturas de dados*, as quais estão listadas na primeira coluna da Tabela 3. O segundo é a dificuldade, determinada pelo percentual de times que resolveu o problema: fácil (≥ 30%), médio (8 a 30%), difícil (2 a 8%) e muito difícil (< 2%). O terceiro é o nível da técnica ou algoritmo principal esperado para solução, de acordo com a presença do tópico na grade curricular e, caso presente, em que momento de um curso de graduação na área de computação o assunto é abordado. São três categorias: *curricular* (núcleo da graduação, como BFS/DFS, Dijkstra, DSU e DP básica), *de fronteira* (eletivas avançadas, como fluxo/*matching*, *segment tree*, KMP e DP de *bitmask*) e *extracurricular*, como *convex hull trick*, decomposição centroide, autômato de sufixos e *sqrt decomposition*. Deste universo, analisou-se as participações da UnB no período, sendo 13 times do Darcy e 7 da FCTE.

3. Resultados

A Tabela 1 resume cada edição, realçando o desempenho do time de melhor classificação de cada campus da UnB. Os melhores resultados vieram do Darcy: 8 problemas e 6º lugar no Brasil em 2022, e 7 problemas com os 6º e 7º lugares em 2025, anos em que a UnB classificou-se para a Final Mundial do ICPC, com os times FLAMENGO e É só fazer, respectivamente. Vale destacar que, em 2025, outros dois times do Darcy avançaram ao *Latin America Championship*: Los Tralalelitos dizem FLAMENGOOO, pela classificação geral, e grafo de botas, como melhor time feminino do continente. As médias de problemas resolvidos por time foram de 5,2 e 4,0 para os campus Darcy Riberiro e FCTE, respectivamente.

Ano	Times	Problemas	Acertos (Rank)		
			Campeão	Darcy	FCTE
2020	302	14	10	6 (23º)	5 (26º)
2021	273	13	8	6 (13º)	4 (27º)
2022	329	13	11	**8 (6º)**	2 (40º)
2023	424	13	11	7 (11º)	4 (38º)
2024	516	12	9	4 (18º)	4 (17º)
2025	593	12	10	**7 (6º)**	5 (20º)

TABELA 1: ESTATÍSTICAS DAS EDIÇÕES DE 2020 A 2025 DA FINAL NACIONAL. AS COLUNAS DARCY E FCTE SE REFEREM AOS TIMES DE MELHOR DESEMPENHO DE CADA CAMPUS

3.1. Dificuldade Extracurricular

A análise do nível da técnica aponta uma forte correlação com a dificuldade de resolução do problema. A Tabela 2 indica o desempenho dos times da UnB em relação aos 77 problemas analisados. Dos 23 problemas curriculares, 21 foram classificados como fáceis ou médios; dos 20 de fronteira, 9 foram médios; e dos 34 extracurriculares, 33 foram difíceis ou muito difíceis, sendo 25 muito difíceis. Neste contexto, o desempenho da UnB está alinhado a esta categorização, uma vez que seus melhores times resolvem uma parte significativa dos problemas que fazem parte do currículo de seus cursos de graduação, porém a taxa de acerto de problemas extracurriculares é baixa.

[1] Placares obtidos em https://scorelatam.naquadah.com.br/.
[2] Dados, *scripts* de análise e código-fonte deste artigo estão disponíveis em https://github.com/bcribas/desempenho-times-maratona.

Nível	Problemas	Dificuldade	UnB
Curricular	23	[muito] fácil	82%
Fronteira	20	médio	46%
Extracurricular	34	[muito] difícil	4%

TABELA 2: TAXA DE RESOLUÇÃO DE PROBLEMAS DA UNB POR NÍVEL DE DIFICULDADE.

Evidencia-se uma barreira de dificuldade: das 96 resoluções da UnB no contexto analisado, 92% foram de problemas fáceis ou médios. Em seis edições, apenas uma solução da UnB para um problema muito difícil foi aceita. O problema em questão foi o *Fascinating Partitions*, que fez parte do caderno da edição 2020, cuja técnica necessária para a solução era programação dinâmica com otimização *Divide and Conquer/Aliens Trick*), tópicos abordados apenas em contextos de fronteira ou extracurriclares. Apenas 6 dentre os 302 times participantes foram capazes de solucionar este problema.

Em 2024, de forma atípica, todos os seis times da UnB resolveram exatamente os quatro problemas mais fáceis (A, F, J e K) e nada além deles. Esse desempenho refletiu uma tendência geral daquela edição: uma expressiva parcela dos participantes ficou no mesmo patamar, com 39 das 63 equipes da competição (61,9%) encerrando o torneio empatadas com esse mesmo conjunto de problemas aceitos. Nesta ocasião, problemas considerados acessíveis no contexto curricular não foram solucionados pela maioria dos times, contrariando a correlação dos demais anos e levando a discussões diversas sobre dificuldade de problemas entre os participantes.

Os times de melhor desempenho da UnB quase sempre acertam as questões fáceis, com algumas exceções que merecem destaque. Em 2022, um time não acertou o problema *Empty Squares*, cuja solução consistia em preencher um tabuleiro com um algoritmo guloso; em 2024, o problema *Jigsaw of Shadows*, que consistia na interseção de intervalos, também não foi resolvido por um time da UnB. Um caso notável é o de 2020: o melhor time da UnB resolveu um problema muito difícil de programação dinâmica (*Fascinating Partitions*), mas não acertou três problemas médios, dois de estruturas de dados (*Excellent Views* e *Job Allocator*) e um de grafos (*Halting Wolf*).

3.2. Abordagem da Solução

A Tabela 3 mostra o desempenho da UnB por tema de problemas. Percebe-se que a UnB tem bom desempenho em temas curriculares, como ad-hoc (86%), guloso e grafos (44%), e ruim justamente em problemas cujas soluções exigem técnicas extracurriculares. A categoria Árvores é um caso limítrofe no contexto analisado, sem registro de acertos da UnB em seus 17 problemas. Pode-se inferir que isso ocorreu porque todos estes problemas exigiam algum conteúdo extracurricular em suas soluções (por exemplo, *rerooting*, decomposição centroide ou isomorfismo).

Categoria (#problemas)	Desempenho		
	Acertos	Submissões	%
Ad-hoc/Implementação	25	29	86%
Strings	6	9	67%
Guloso	16	31	52%
Grafos	11	25	44%
Construtivo/Matemática	11	28	39%
Geometria	8	25	32%
Programação dinâmica	4	13	31%
Fluxo/*Matching*	5	22	23%
Combinatória/Probabilidade	2	13	15%
Estruturas de dados	*5*	*37*	*14%*
Árvores	*0*	*17*	*0%*

TABELA 3: DESEMPENHO DA UNB POR TEMA

O tema *estruturas de dados* (14%) merece detalhamento, dada a sua amplitude. As estruturas de dados com maior número de acertos pela UnB estão na categoria *de fronteira*, como *sparse table*, pilha monotônica e *segment tree* com *lazy propagation*, responsáveis pelas poucas resoluções do período analisado: *Excellent Views* em 2020 e *Emergency Rations* em 2025. Nota-se a ausência de acertos em um conjunto bem definido de estruturas avançadas: *convex hull trick* e *Li Chao tree*, varreduras *offline* com *segment tree* ou Fenwick, *sqrt decomposition* e combinações de RMQ com busca binária. A única estrutura considerada difícil que a UnB aplicou com sucesso foi a comparação de vetores de frequência por *hashing* em *Candy Rush* (2023), e somente por um time do Campus Darcy.

3.3. Darcy vs FCTE

Frente aos times brasileiros melhor colocados, a lacuna perante a UnB se concentra nas categorias de nível extracurricular. Ela é inferida da própria composição das resoluções da instituição, cruzada com a dificuldade de cada problema: as categorias em que a UnB quase não pontua — estruturas de dados avançadas (14%), árvores (0%), grafos difíceis e geometria — são exatamente as dominadas por técnicas extracurriculares, que definem os problemas que separam os primeiros colocados do meio da tabela[3]. Para superá-la, os dados mostram dois caminhos.

O primeiro é a profundidade: nos anos onde times da UnB avançaram para etapas internacionais, o time principal do Darcy acertou um ou dois problemas de nível extracurricular. Foram apenas 5 resoluções extracurriculares em seis anos, todas de times do Darcy, sobre quatro problemas distintos: *Fascinating Partitions* (2020, programação dinâmica), *Lazy Printing* (2022, *strings*, por dois times), *Candy Rush* (2023, estruturas de dados) e *Health in Hazard* (2023, geometria). Ter, num ano, um time capaz de solucionar um problema extracurricular é o que distingue as edições em que a UnB teve melhor desempenho.

[3] A base analisada registra os problemas resolvidos por cada time da UnB e a dificuldade global de cada problema; não inclui, problema a problema, o que cada time concorrente resolveu.

O segundo é a amplitude: em 2025, o time É só fazer chegou à final mundial resolvendo sete problemas nos níveis curriculares e de fronteira com rapidez e poucas falhas. Isso indica que a consistência nestes níveis pode ser suficiente para avançar nas etapas, embora ainda esteja claro que há espaço para melhorar o desempenho quando comparado aos times das primeiras colocações.

Os dois *campi* têm perfis distintos. O campus Darcy resolve mais problemas e de assuntos mais abrangentes, cobrindo programação dinâmica (4 acertos contra nenhum da FCTE), *strings* (5 contra 1) e grafos (9 contra 2). A FCTE não resolveu nenhum dos 41 problemas extracurriculares do intervalo analisado, concentrando-se nas categorias *ad-hoc* e guloso, de modo que sua barreira de dificuldade começa antes da barreira do campus Darcy Ribeiro. A Figura 1 contrasta a composição das resoluções de cada *campus* por nível de ensino: o Darcy distribui 68 resoluções entre curriculares (63%), de fronteira (29%) e extracurriculares (7%), enquanto as 28 resoluções da FCTE se restringem aos níveis curricular (79%) e de fronteira (21%), sem nenhuma resolução extracurricular.

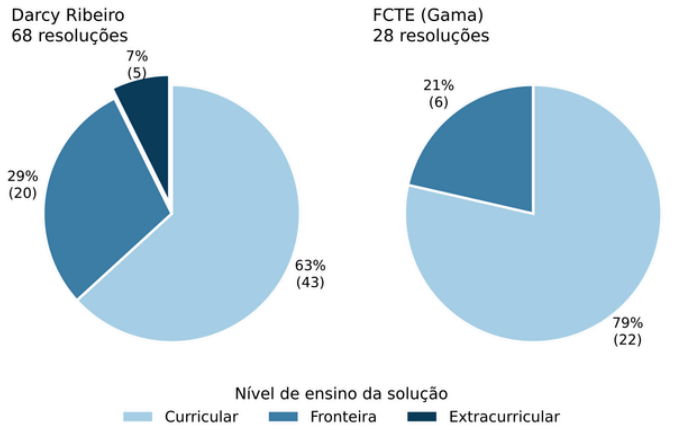


FIGURA 1: RESOLUÇÕES DE CADA CAMPUS POR NÍVEL DE TÉCNICA DA SOLUÇÃO

4. Discussão e Recomendações

O diagnóstico parece claro: o gargalo não está nos problemas de nível fácil, que os times resolvem com eficiência. A verdadeira barreira está na profundidade extracurricular necessária para avançar do nível intermediário para o topo — algo que, por não constar no currículo regular, exige um treinamento dedicado.

Visando um melhor desempenho, os dados indicam que a UnB deve priorizar as estruturas de dados além do conteúdo básico (*segment tree* com *lazy*, *convex hull trick* e *sqrt decomposition*), os algoritmos em árvores (*rerooting*, decomposição centroide), de fluxo e de *matching*, e geometria computacional. Para a FCTE, a prioridade é anterior, buscar o domínio de fronteira em grafos e em DP de nível médio.

Uma vantagem da UnB é a sinergia do treinamento inter-campi. Embora haja a competitividade, Darcy e FCTE têm trabalhado em conjunto no desenvolvimento dos competidores. Este contexto pode ser aproveitado para que o Darcy alavanque a FCTE, como já ocorreu de forma inversa no passado.

Transformar em rotina o acerto extracurricular isolado, que o Darcy já produz em anos bons, é o que pode levar a UnB de visitante ocasional a presença constante na Final Mundial.

5. Conclusão

Em seis edições, os times da UnB exibem um perfil consistente e explicável: resolvem com eficiência o que a graduação ensina (82% dos problemas curriculares) e quase nunca o que ela não ensina (4% dos extracurriculares). A lacuna tem nome (estruturas de dados avançadas, árvores, grafos difíceis e geometria) e aponta de forma objetiva para o que treinar. O campus Darcy lidera a corrida no período avaliado, e o campus FCTE precisa primeiro ampliar seu repertório.

O método, baseado só em placares públicos e enunciados, é replicável a cada edição como painel de acompanhamento dos times, e os *scripts* de coleta e análise estão disponíveis no repositório do artigo. Como trabalho futuro, pretende-se generalizar a extração para receber apenas o nome de uma instituição (por exemplo, "USP") e gerar automaticamente o mesmo diagnóstico para todos os seus *campi*.

A análise proposta levanta diversos pontos relevantes e viabiliza um direcionamento para ações imediatas. Ainda assim, uma avaliação mais minuciosa dos dados, inclusive com métricas mais objetivas, pode levar a novas percepções para as instituições que buscam melhorar seus desempenhos.

Referências

[1] Naquadah. Placares da Maratona SBC / ICPC Latin America. Disponível em: https://scorelatam.naquadah.com.br/. Acesso em: jun. 2026.
[2] Sociedade Brasileira de Computação. Maratona SBC de Programação. Disponível em: https://maratona.sbc.org.br/. Acesso em: jun. 2026.
[3] ICPC Foundation. International Collegiate Programming Contest — Latin America. Disponível em: https://icpc.global/. Acesso em: jun. 2026.
[4] Cassio Polpo de Campos and Carlos Eduardo Ferreira. BOCA: um sistema de apoio a competições de programação. In: Workshop de Educação em Computação e Informática do Estado de Minas Gerais (WEIMASC), 2004.



DIAMANTE

VTEX

Problemas à altura do talento brasileiro

A VTEX, como empresa de origem brasileira, sempre acreditou no Brasil como um celeiro de talentos excepcionais, capazes de gerar impacto real em escala global. Parte do nosso papel é conectar essas pessoas a problemas complexos e ainda sem solução.

Estamos ao lado da Maratona de Programação desde 2018, acompanhando suas diferentes fases e gerações de competidores. A Maratona reúne gente com uma capacidade rara, e que admiramos: enfrentar problemas difíceis, aprender rápido e construir soluções onde a resposta não é óbvia. Quando essa capacidade encontra desafios reais, em escala como no nosso caso os de transformar uma indústria inteira como a de digital commerce, o resultado aparece.

Não é por acaso que nossa relação com a Maratona atravessou tantos anos. Muitos dos competidores que conhecemos nesse ambiente passaram a fazer parte da VTEX e chegaram a papéis de referência e liderança, em um percurso que começou muito antes de qualquer processo seletivo. É uma relação de longo prazo com uma comunidade da qual temos orgulho de estar próximos.

Neste 31° ano da Maratona de Programação da SBC, os maratonistas desta edição também passam a fazer parte dessa história. A competição termina, mas a comunidade vai muito além dela. E, nos próximos problemas difíceis que essa geração escolher resolver, esperamos continuar por perto.

(foto) *Registro da Primeira Fase de 2019 realizada no escritório da VTEX no Rio de Janeiro.*

BRASIL, O PAÍS DA PROGRAMAÇÃO COMPETITIVA?

Em duas das principais olimpíadas internacionais de informática, as delegações brasileiras alcançaram resultados históricos: lideraram o ranking de medalhas da EGOI e conquistaram o quarto ouro do país na IOI.

Na EGOI, realizada em Cesenatico, na Itália, as quatro representantes brasileiras conquistaram medalhas. Maria Clara Fontes Silva foi ouro e terminou em segundo lugar na classificação geral; Sofia Torres de Paula Cintra também recebeu ouro; Julia Galdino Tiosso Lopez e Talita Ribeiro de Rezende ganharam prata. Com dois ouros e duas pratas, o Brasil liderou o ranking de medalhas de uma edição que reuniu 248 competidoras de 69 países.

Meses depois, na IOI de Tashkent, no Uzbequistão, todos os integrantes da equipe brasileira também foram premiados. Pedro Suyama Leston Rey conquistou ouro; João Paulo Campos de Oliveira, prata; e Maria Clara Fontes Silva e Enzo Ivamoto Hirota, bronze. Foi apenas o quarto ouro do Brasil desde sua estreia na IOI, em 1999. Nesta edição, com 375 competidores de 92 países, o desempenho brasileiro superou o de países como Canadá, Alemanha, França e Inglaterra no quadro de medalhas.

As conquistas também colocam em evidência o trabalho contínuo da OBI, das escolas, dos professores e de todos os envolvidos na preparação e no acompanhamento das delegações.

“Os resultados são por mérito dos alunos. Eles se dedicam durante anos a aprofundar seus conhecimentos em programação, matemática e algoritmos. Eles são treinados por seus professores nas escolas e também por uma rede de ex-alunos da OBI, que é também dedicada e entusiasmada (muitos deles competidores da Maratona enquanto alunos da graduação). É muito trabalho envolvido. Mas também é gratificante saber que a OBI tem conseguido estimular e ajudar a treinar um grande grupo de alunos, e também de selecionar os melhores alunos para as competições internacionais.” - Professor Ricardo Anido

O país está ocupando posições de destaque nas maiores competições internacionais para estudantes da educação básica. Quanto tempo falta para vermos uma equipe brasileira recebendo uma medalha nas Finais Mundiais do ICPC novamente?





31ª MARATONA SBC
{DE PROGRAMAÇÃO 2026}

FINAL BRASILEIRA EM UBERLÂNDIA:
05 A 08 DE NOVEMBRO DE 2026

PRIMEIRA FASE:
29 DE AGOSTO DE 2026





REALIZAÇÃO

SBC Sociedade Brasileira de Computação

PROMOÇÃO

Maratona de Programação

APOIO

CNPq CAPES Facom

UFU FAPEMIG

PATROCINADORES

GLOBAL

JETBRAINS

PREMIUM

btg pactual

DIAMANTE

Pax VTEX INCOGNIA nu rivio

PRATA

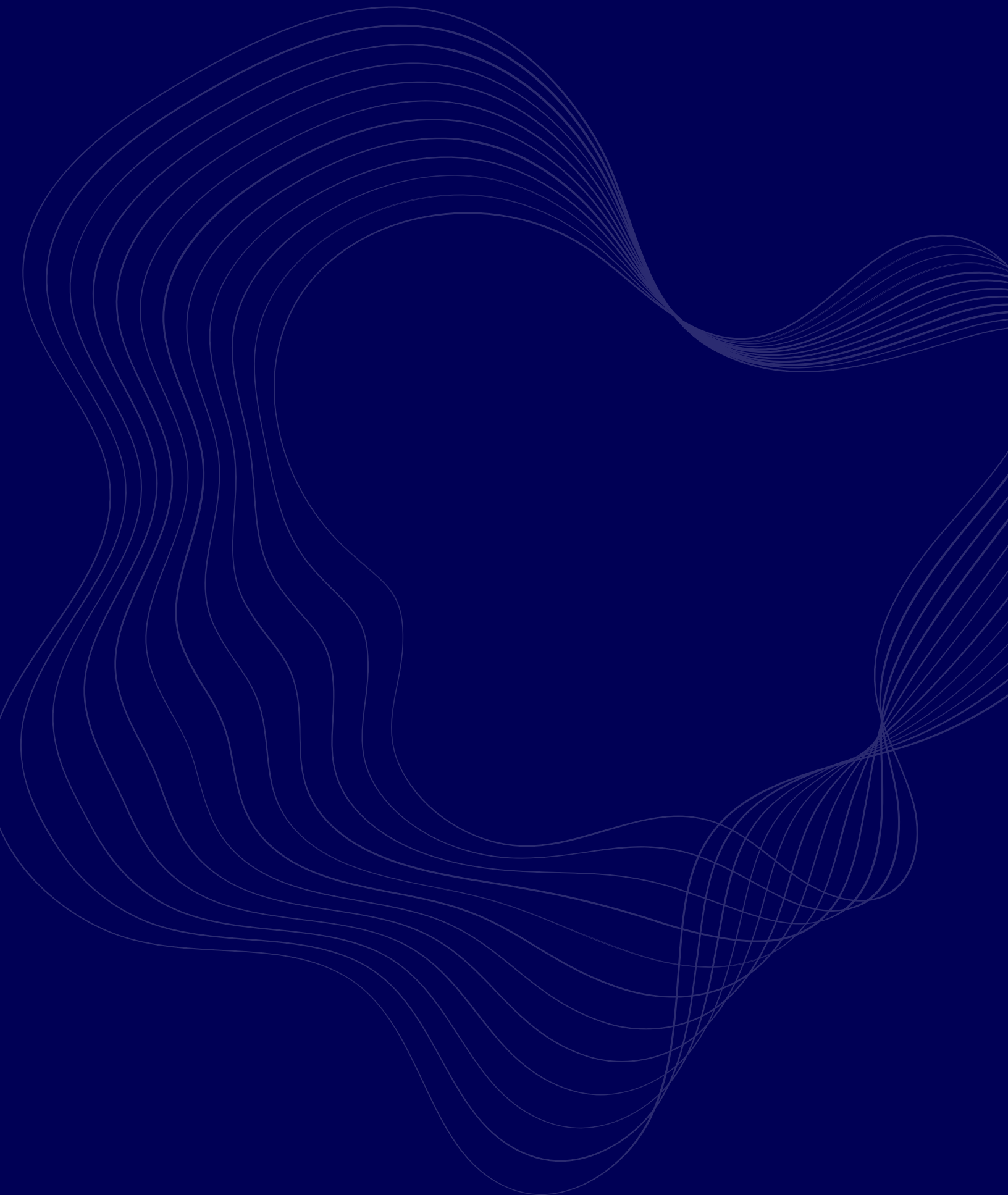
HUAWEI stone

BRONZE

ALPHABOT Google Mendelics Análise genômica

NUVEM

UNDER SERVERS & DATACENTERS





FACEBOOK.COM/MARATONA

INSTAGRAM.COM/MARATONADEPROGRAMACAO

X.COM/MARATONAPROG

LINKEDIN.COM/COMPANY/MARATONA-DE-PROGRAMACAO

YOUTUBE.COM/@MARATONASBC