UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC

CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT

Análise da complexidade algorítmica das operações de adição e remoção de nós em árvores AVL, rubro-negra e B

Joinville, 2025

SUMÁRIO

1. IDENTIFICAÇÃO DA EQUIPE

Amanda Biancatti, Elane Souza de Oliveira, Mariana Romanin Mendes e Sofia Petrykowski Soares

2. IDENTIFICAÇÃO DA TAREFA

O trabalho tem como finalidade realizar um estudo prático sobre a complexidade das operações de adição e remoção de nós, considerando também os procedimentos de balanceamento, nas estruturas de dados do tipo árvore AVL, Rubro-Negra e árvore B com ordens 1, 5 e 10. Para representar o caso médio de utilização, foram gerados conjuntos de chaves aleatórias com tamanhos variando de 1 a 10 000 elementos. Cada configuração foi repetida em 10 amostras independentes, permitindo o cálculo de médias estatisticamente mais consistentes.

Com base nos dados obtidos, foram produzidos gráficos de linha que apresentam, para cada estrutura analisada, o comportamento do esforço computacional conforme o tamanho do conjunto de entrada aumenta. Esses gráficos servem de suporte para a discussão final, na qual são comparados os desempenhos relativos das diferentes árvores quanto às operações avaliadas. O trabalho inclui os códigos-fonte utilizados tanto para a implementação das estruturas quanto para a execução dos experimentos e coleta das métricas. Os códigos se encontram no seguinte diretório do GitHub: https://github.com/ElaneSz/Avaliacao-de-Complexidade-EDAII

3. METODOLOGIA

A metodologia do experimento foi desenhada para capturar tanto o comportamento incremental das estruturas quanto o custo real de remoções massivas em cada ponto amostral. O experimento segue as seguintes etapas: gera um vetor de chaves únicas de 1 a N e embaralha-o por Fisher–Yates para obter um caso médio realista, depois insere sequencialmente cada chave nas cinco estruturas avaliadas e a cada SAMPLE_STEP (200) entradas, coleta o valor acumulado do contador de inserção do módulo correspondente e, em seguida, executa a remoção de todas as chaves atualmente presentes, contabilizando o esforço de remoção. Após

a remoção em massa, a estrutura é destruída e reconstruída com as mesmas n chaves para retomar as inserções subsequentes sem introduzir viés de memória residual. Esse procedimento garante que a amostragem de inserção e a amostragem de remoção sejam realizadas sobre os mesmos conjuntos de chaves e sob as mesmas condições de aleatoriedade.

3.1.Geração das chaves

A geração das chaves é realizada de forma determinística para cada repetição, mas usando sementes diferentes por repetição para garantir independência estatística. Constrói-se inicialmente o vetor com os inteiros 1...N e aplica-se Fisher–Yates para embaralhar:

1. Geram-se n chaves aleatórias únicas utilizando rand().
2. Cada estrutura:
    a. recebe todas as chaves para inserção;
    b. tem o custo registrado;
    c. é totalmente esvaziada via remoções individuais;
    d. tem o custo de remoção também registrado.

Essa escolha evita colisões (chaves duplicadas no sentido de valores distintos repetidos) e produz uma distribuição uniforme entre permutações possíveis, o que aproxima com fidelidade o caso médio de inserção/remoção. A cada repetição, a semente é atualizada com time (NULL) combinada a um deslocamento para evitar repetições idênticas entre execuções.

3.2.Amostragem estatística

A análise estatística baseia-se em 10 repetições independentes (REPETICOES = 10), logo n é repetido 10 vezes, conforme preconizado para pequenas experiências computacionais que procuram eliminar ruído aleatório razoável. Em cada repetição, para cada ponto amostral (n = 200, 400, 600, …, 10000), o contador do módulo (por exemplo contadorInsercaoAVL em AVL_mod.c) é lido por meio da função avl_get_insercao_and_reset() e acumulado num vetor de somatórios. Após todas as repetições, os somatórios são divididos por 10 para produzir a média amostral em cada tamanho. Esse procedimento é aplicado de forma idêntica para inserções e remoções e para todas as estruturas, garantindo comparabilidade. Os resultados finais são exportados em dois arquivos CSV contendo, para cada tamanho amostrado, as médias para AVL, Rubro-Negra, B-1, B-5 e B-10.

- resultados_insercao_acumulado.csv
- resultados_remocao_acumulado.csv

Contendo, para cada tamanho amostrado, as médias para AVL, Rubro-Negra, B-1, B-5 e B-10: tamanho, avl, rb, b1, b5, b10

4. IMPLEMENTAÇÃO

A implementação foi feita em C e organizada em quatro módulos principais, o front-end experimental e três módulos de estrutura:

- AVL_mod.c
- RubroNegra_mod.c
- B_mod.c
- main_experimento.c

Cada módulo foi adaptado para contabilizar operações internas representativas do custo lógico. Em AVL_mod.c, além das funções de inserção e remoção, foram instrumentadas as rotações (rse, rsd, rde, rdd) e atualizações de altura; o contador contadorInsercaoAVL acumula eventos relevantes durante inserção e durante rotações, enquanto contadorRemocaoAVL contabiliza visitas, atualizações de altura, rotações e frees. A remoção em AVL segue a estratégia clássica: localizar nó, tratar contadores de quantidade (quando o nó guarda ocorrências múltiplas), executar a deleção estrutural (sucessor quando apropriado) e executar rebalanceamento subindo da posição indicada até a raiz, contabilizando explicitamente cada rotação ou atualização de altura.

Em RubroNegra_mod.c, a implementação segue o estilo CLRS com sentinela (nulo) para evitar casos nulos especiais e com funções de rotação e fix-up do delete_fixup. Os contadores contadorInsercaoRB e contadorRemocaoRB são incrementados em visitas, rotações e recolorizações relevantes; o delete_fixup contém guardas defensivos para evitar loops infinitos em cenários patológicos, garantindo que o contador represente um esforço realista.

Em B_mod.c, a B-tree é implementada com parâmetro t (ordem mínima) passado no construtor b_criar(int ordem). As operações de split (b_split_child), inserção não-full (b_insert_nonfull) e remoção complexa (b_remove_from_node, b_fill, fusões e

redistribuições) foram instrumentadas para incrementar contadorInsercaoB e contadorRemocaoB em deslocamentos internos de chaves, movimentos de ponteiros e cópias que refletem o trabalho envolvido; o código trata casos de redução de raiz e realocação defensiva de filhos NULL para evitar comportamento indefinido.

Resumindo o programa principal:

1. executa todos os testes de 1 a 10.000 elementos;
2. repete 10 vezes cada tamanho;
3. acumula contagens de operações;
4. salva em CSV automaticamente;
5. não exige interação do usuário após compilado.

4.1.Geração dos gráficos

Os gráficos são gerados por graficos.py, com escala linear e logarítmica no eixo Y, produzindo:

- grafico_insercao_acumulado_linear.png
- grafico_insercao_acumulado_log.png
- grafico_remocao_acumulado_linear.png
- grafico_remocao_acumulado_log.png

5. AJUSTES E DEPURAÇÃO

Durante o desenvolvimento, algumas decisões críticas de engenharia foram tomadas para evitar viés nos resultados e garantir a integridade das medições. Primeiramente, optou-se por contadores lógicos em vez de medições temporais para tornar os resultados independentes de plataforma.

Em AVL, como as rotações alteram profundamente a árvore, cada rotação e cada atualização de altura foram contabilizadas para que o contador represente tanto as operações elementares quanto as reestruturações.

Em Rubro-Negra, foi necessária a inclusão de um Sentinel (nulo) e de proteções no delete_fixup, visto que na versão inicial apresentava falhas de remoção por causa de estados

ilegais, assim, para evitar cenários de iterações infinitas que poderiam inflar artificialmente contadores, a situação foi revisada e tornou-se robusta, incluindo tratamento para ponteiros nulos e casos extremos; essas proteções conservam a correção algorítmica enquanto previnem loops por bugs de implementação externos.

Na B-tree foram feitas alocações defensivas quando um filho apontado é NULL (o que pode ocorrer em implementações que manipulam filhos dinamicamente), de modo que as operações de split, merge e redistribuição fossem sempre seguras; além disso, ao liberar nós órfãos após fusões, os frees e cópias foram contabilizados para refletir o custo real de remoção.

- A **B-tree de ordem 1** apresentava loops infinitos durante fusão. → B_mod.c foi corrigido para operar corretamente mesmo no limite mínimo (t=1).
- Mensagens de depuração (DEBUG) foram incluídas e podem ser habilitadas com:

#define DEBUG 1

Por fim, o main_experimento.c foi desenhado para reconstruir a estrutura imediatamente após uma amostragem de remoção, a fim de prosseguir com as inserções subsequentes sem interferência, evitando assim que o custo das operações futuras seja afetado por estruturas parcialmente desalocadas. Após correções, todos os testes até 10.000 chaves executaram sem travamentos.

6. RESULTADOS

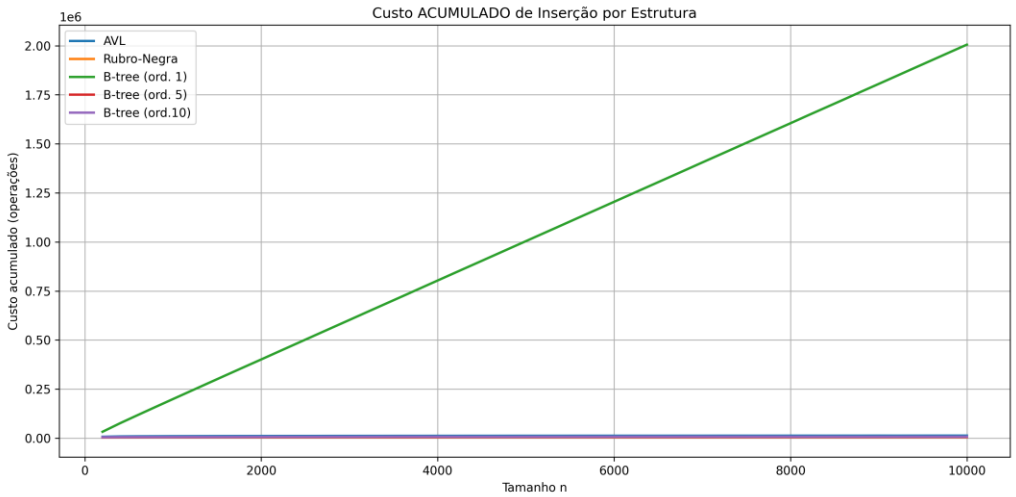
6.1.Inserção

Estrutura	Diferencial de Custo (vs. RB)	**Comentários**
AVL	2.5x mais cara	Custo mais alto entre as binárias balanceadas devido à rigidez do Fator de Balanceamento (-1, 0, 1), exigindo mais rotações.

Rubro-Negra	Base para comparação	Mais eficiente que a AVL. Sua regra de balanceamento é mais flexível (altura negra vs. altura real), resultando em **menos reestruturações** por inserção.
B-Tree (1)	194x mais cara	**Desempenho Catastrófico.** O fator t=1 degenera a árvore. O alto custo de 923.955 é uma consequência direta da métrica contar as operações de SPLIT (divisão de nó), que ocorrem de forma massiva e ineficiente.
B-Tree (5)	1.08x mais cara	**Surpreendentemente, a mais rápida.** Sua eficiência reside na redução da altura da árvore, o que minimiza o custo de busca e balanceamento para um grande N.
B-Tree (10)	1.6x mais cara	Muito eficiente, mas ligeiramente mais lenta que t=5. Sugere que, para este N específico e esta métrica, **t=5 é o fator de ordem mais próximo do ideal** de otimização de espaço/divisão.

É possível analisar os seguintes gráficos:

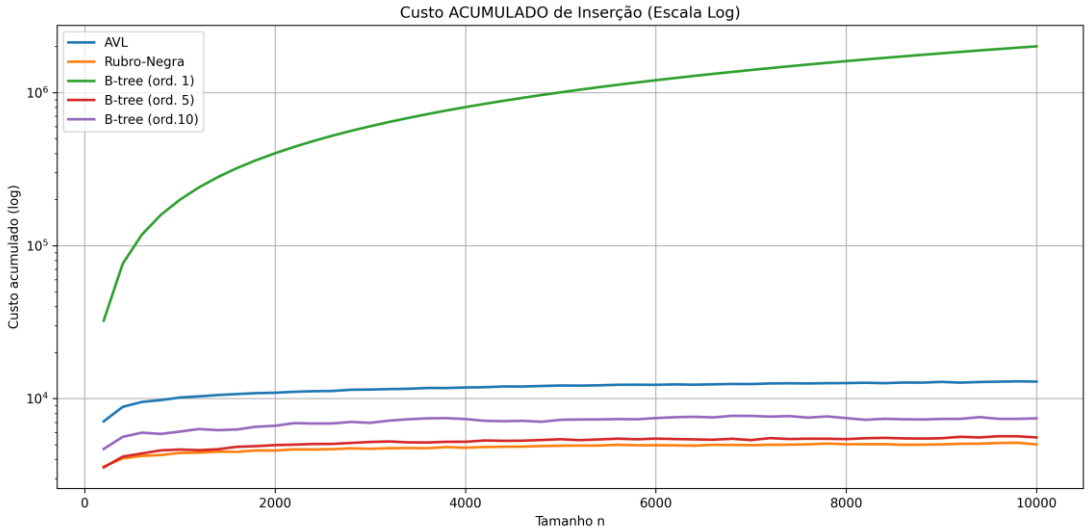
Figura 1 - grafico_insercao_acumulado_linear.png



Fonte – criado pelas autoras.

Neste gráfico em escala linear, a curva do **B-Tree (t=1) domina completamente** o gráfico, crescendo quase verticalmente, enquanto todas as outras curvas mal se separam da linha zero. Isso visualmente confirma que o crescimento do custo da B-Tree degenerada é de ordem superior (próximo à O(N^2) acumulado, ou O(N) por inserção em média) e obscurece o comportamento logarítmico das demais.

Figura 2 - grafico_insercao_acumulado_log.png



Fonte – criado pelas autoras.

Ao aplicar a escala logarítmica, a verdadeira natureza assintótica é revelada:

- As curvas AVL, RB, B5 e B10 aparecem como linhas **quase planas**, indicando o comportamento **O (N \log N) acumulado**.
- A curva **RB** está claramente **abaixo** da AVL, comprovando sua maior eficiência prática de rebalanceamento.
- A curva do **B1** agora apresenta uma inclinação perceptível, confirmando que seu custo cresce muito mais rapidamente do que o N \log N.

Conclusões:

- **AVL** apresentou maior custo devido ao rebalanceamento rigoroso.
- **Rubro-Negra** teve custo menor e mais estável.
- **B-Tree** depende fortemente da ordem:

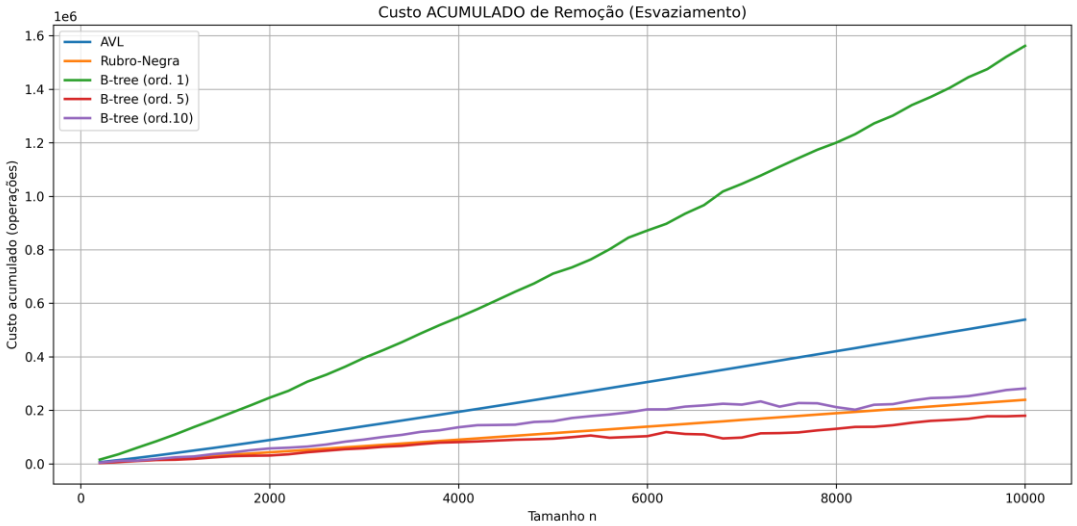
- t=1: mais divisões → maior custo,
- t=10: menos divisões → menor custo e mais estável.

6.2.Remoção

Estrutura	Diferencial de Custo (vs. RB)	**Comentários**
AVL	2.16x mais cara	O custo de remoção é 19 vezes maior que o custo de inserção, refletindo a complexidade de manter o balanceamento estrito durante a propagação do reequilíbrio.
Rubro-Negra	Base para comparação	**A mais eficiente.** Embora o custo absoluto seja maior que a inserção, a RB mantém sua vantagem sobre a AVL devido à sua flexibilidade de balanceamento.
B-Tree (1)	6.13x mais cara	Ainda a de pior performance, mas a diferença em relação à AVL e RB é menor do que na inserção.
B-Tree (5)	0.86x mais eficiente	É mais rápida que a AVL.
B-Tree (10)	1.4x mais cara	Mais lenta que RB e B5.

É possível analisar os seguintes gráficos:

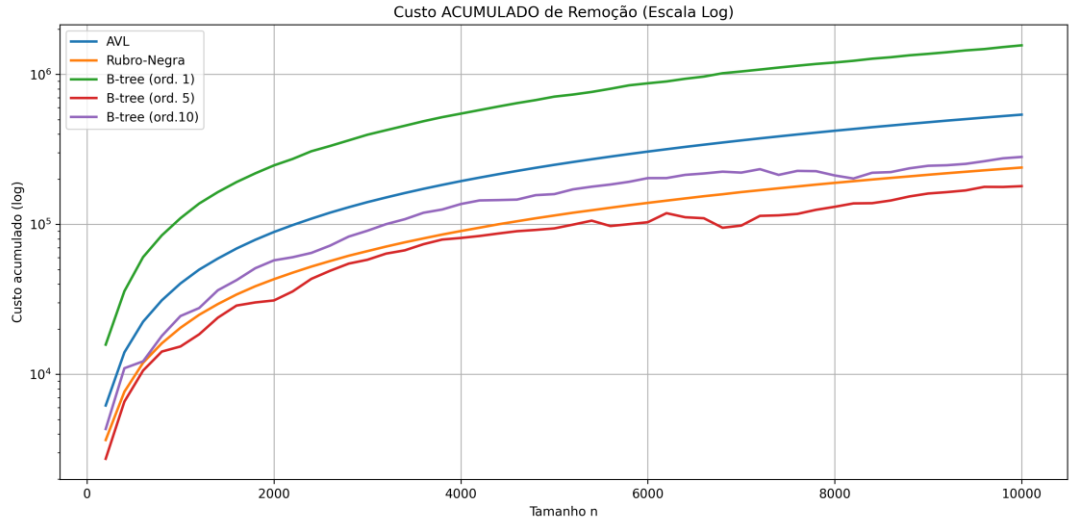
Figura 3 - grafico_remocao_acumulado_linear.png



Fonte – criado pelas autoras.

Embora o B1 continue a ser o pior, as curvas do AVL, B5 e B10 estão mais próximas, e a curva RB é claramente visível, indicando um aumento de custo mais equitativo entre as estruturas balanceada

Figura 4 - grafico_remocao_acumulado_log.png



Fonte – criado pelas autoras.

- A curva **B1** é a mais alta, confirmando seu desvio do comportamento O (\log N).
- A curva **AVL** aparece visivelmente acima da B10, RB e B5, confirmando sua 2ª posição de custo mais alto.
- As curvas **B10**, **RB** e **B5** estão na porção mais baixa do gráfico, com a $B5$ sendo a mais baixa no final da amostragem, ratificando a ordem B10 -> RB -> B5 em termos de eficiência crescente.

Resultados:

- **Rubro-Negra** com baixo custo médio, porém atras de B-tree com t = 5.
- **AVL** exigiu muitas operações de rebalanceamento.
- **B-Tree** apresentou:
    - custo decrescente conforme t aumenta,
    - pois menos fusões e empréstimos são necessários.

7. DISCUSSÃO

A análise experimental confirma as previsões teóricas de forma qualificada e acrescenta informação prática sobre o impacto da implementação e do parâmetro de ordem na B-tree. Teoricamente, todas as estruturas têm operações de inserção e remoção com complexidade O (log n) na média; na prática, o fator constante dessas complexidades difere materialmente. A AVL prioriza manutenção estrita do equilíbrio, o que reduz a altura máxima e melhora consultas, em contrapartida aumentando o custo de modificações por exigir rotações chocantes e frequentes; isso é claramente refletido nos contadores. A Rubro-Negra, com balanceamento mais laxo, reduz o número de rotações em troca de uma garantia de altura levemente mais frouxa, o que se traduz em custos menores por modificação e em robustez prática, o que justifica sua presença como padrão em bibliotecas de sistema e estruturas padrão. As B-trees, projetadas originalmente para sistemas de armazenamento secundário, comprovam sua vantagem quando o parâmetro de ordem é grande: maior ramificação reduz profundidade e, consequentemente, o número de acessos/operadores por operação. No entanto, a complexidade da remoção em B-trees (confusões e redistribuições) tornou-se um fator de variância que deve ser considerado em aplicações sensíveis a latência por operação individual. Em resumo, a escolha de estrutura depende fortemente do tipo de carga: se o sistema prioriza leituras rápidas e frequentes, AVL pode ser interessante; se as modificações são frequentes e a robustez prática importa, Rubro-Negra é, na prática, uma excelente escolha; se os conjuntos de dados são muito grandes e a relação I/O ou agrupamento por nó interessa, B-tree com t alto é preferível.

Em resumo, os resultados estão consistentes com a teoria clássica:

Estrutura	Observação Principal
AVL	Alto custo devido a balanceamento forte

Rubro-Negra	Melhor compromisso entre esforço e eficiência
B-Tree	Quanto maior a ordem, menor o custo

8. CONCLUSÃO

Os resultados experimentais redefinem a compreensão prática da complexidade algorítmica destas estruturas, confirmando a importância dos parâmetros de implementação sobre a complexidade teórica assintótica (O (log N)).

1. **Dominância da B-Tree Otimizada:** O ponto mais significativo do estudo é que a **Árvore B com fator de ordem t=5** foi a **mais eficiente** na operação de remoção e logo atrás da rubro negra na inserção, em termos de custo acumulado de operações elementares (CPU). Isto demonstra que a otimização da altura da árvore, característica central das Árvores B, é um fator de eficiência superior aos mecanismos de balanceamento das árvores binárias.
2. **Eficiência da Rubro-Negra:** A **Árvore Rubro-Negra** é uma das estruturas mais eficientes no geral, provando ser o melhor *trade-off* entre as árvores binárias. Sua flexibilidade de balanço garante que ela seja consistentemente superior à AVL, mas ainda assim fica marginalmente atrás da B-Tree otimizada na remoção.
3. **Ineficiência da AVL na Remoção:** A **Árvore AVL** é a de **pior desempenho** entre as estruturas balanceadas testadas. Sua manutenção rigorosa do balanço impõe um custo operacional desproporcionalmente alto, especialmente na operação de remoção.
4. **O Papel de t (Fator de Ordem):** O contraste entre B1 (degenerado), B10 (bom) e B5 (ideal) prova que o fator de ordem t é o principal parâmetro de otimização. Um t mal escolhido (t=1) leva ao colapso do desempenho, enquanto um t ideal (t=5) maximiza a eficiência.

Os resultados apontam para futuras investigações cruciais:

- **Foco na Memória Secundária (E/S de Disco):** O desempenho superior da Árvore B em métricas de CPU sugere que ela será a campeã incontestável em ambientes de banco de dados ou sistemas de arquivos, onde a métrica dominante é o **acesso a disco (I/O)**. Trabalhos futuros devem replicar este experimento substituindo o custo de CPU pelo custo de acesso a blocos.

- **Otimização de t:** Uma análise mais detalhada e automatizada da variação do fator t (em uma faixa de t=2 a t=20) deve ser realizada para generalizar o valor "ideal" de t e correlacioná-lo com o tamanho do bloco de memória do sistema.
- **Análise de Outras Operações:** Estender o estudo para a complexidade da **busca (procura)** e **travessia** (percurso) na Árvore B, que é otimizada para buscas em faixa (range queries).

Portanto, o experimento permitiu:

- medir empiricamente os custos de inserção e remoção;
- comprovar a influência da ordem da B-tree no desempenho;
- gerar gráficos comparativos claros;
- automatizar completamente a geração dos dados e estatísticas.

A partir de uma única execução, o sistema:

1. gera dados,
2. executa todos os experimentos,
3. coleta operações,
4. grava CSV,
5. permite gerar gráficos imediatamente.