UNIVERSIDADE FEDERAL DO TOCANTINS
CÂMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DANIEL VINICIUS DA SILVA, ÉRICK SANTOS MARÇAL, BENEDITO JAIME MELO MORAES JUNIOR

COMPENSAÇÕES DE ESPAÇO E TEMPO

PALMAS (TO)
2024

1 INTRODUÇÃO

No campo da ciência da computação, a análise de trade-offs entre espaço e tempo é uma questão fundamental que influencia diretamente o desempenho e a eficiência dos algoritmos. O conceito de trade-offs entre espaço e tempo refere-se à relação inversa entre a quantidade de memória (espaço) que um algoritmo utiliza e o tempo necessário para executar suas operações. Em termos simples, otimizar um desses recursos frequentemente resulta em um aumento no uso do outro.

Por exemplo, algoritmos que são altamente eficientes em termos de tempo podem consumir grandes quantidades de memória, enquanto algoritmos que utilizam menos memória podem exigir mais tempo para processar os dados. Esta troca é uma consideração crítica ao projetar sistemas de software, onde o balanceamento adequado entre espaço e tempo pode determinar a viabilidade e a escalabilidade de uma solução.

Neste trabalho, exploraremos diversos aspectos dos trade-offs entre espaço e tempo, incluindo suas implicações práticas em diferentes cenários de aplicação. Analisaremos exemplos clássicos e contemporâneos, destacando como esses trade-offs são abordados na prática e as estratégias utilizadas para otimizar o desempenho dos algoritmos. Além disso, discutiremos como avanços em hardware e tecnologia influenciam essas considerações e as tendências futuras nesse campo.

A compreensão e a gestão eficaz dos trade-offs entre espaço e tempo são essenciais para engenheiros de software, desenvolvedores de sistemas e pesquisadores, pois afetam diretamente a eficiência, a responsividade e a escalabilidade das aplicações computacionais. Ao longo deste estudo, procuraremos fornecer uma visão abrangente e detalhada das técnicas e abordagens empregadas para equilibrar essas importantes considerações de recursos.

2 REVISÃO TEÓRICA

2.1 Sorting by Counting

Sorting by Counting é um algoritmo de ordenação estável e não comparativo. Foi introduzido por Harold H. Seward em 1954. Esse algoritmo é eficaz para ordenar listas de inteiros ou objetos cujas chaves são inteiros, onde o intervalo dos valores possíveis é conhecido e relativamente pequeno. Funciona contando o número de ocorrências de cada valor distinto em uma lista e, em seguida, calculando a posição de cada valor na lista ordenada. O algoritmo é eficiente em termos de tempo com complexidade $O(n+k)$, onde n é o número de elementos na lista e k é o valor máximo na lista. No entanto, pode ser ineficiente em termos de espaço para listas com valores altamente dispersos.

2.2 Algoritmo de Força Bruta

O algoritmo de força bruta é uma abordagem direta e simples para resolver problemas de busca de padrões em strings. Também conhecido como busca ingênua, este algoritmo examina todas as possíveis posições do padrão na string alvo, verificando uma por uma se há uma correspondência exata. A simplicidade dessa técnica é sua principal vantagem, mas também sua maior limitação, especialmente para textos grandes ou padrões longos.

Para um texto de comprimento n e um padrão de comprimento m, o algoritmo de força bruta funciona da seguinte maneira:

Alinha o padrão com o início do texto. Compara o padrão com a substring do texto de mesmo comprimento. Se todas as comparações forem verdadeiras, a posição atual é uma ocorrência do padrão. Se não houver correspondência, o padrão é deslocado uma posição à direita e o processo é repetido até que todas as posições possíveis sejam verificadas. A complexidade temporal do algoritmo de força bruta é $O(n \cdot m)$ no pior caso, o que o torna impraticável para grandes entradas. No entanto, sua implementação simples e direta faz com que seja útil em situações onde a eficiência não é uma preocupação crítica ou onde o texto e o padrão são relativamente curtos.

2.3 Algoritmo KMP (Knuth-Morris-Pratt)

O algoritmo KMP, desenvolvido por Donald Knuth, Vaughan Pratt e James H. Morris em 1977, é uma técnica de busca de padrões em strings que melhora a eficiência em comparação com a abordagem de força bruta. O KMP evita comparações redundantes ao pré-processar o padrão para criar uma tabela de "próximos saltos", que indica o quanto o padrão pode ser deslocado sem perder correspondências já verificadas.

A estrutura do algoritmo KMP é a seguinte:

Pré-processamento: Cria-se uma tabela (também conhecida como tabela de falhas) que armazena os comprimentos dos prefixos próprios que são também sufixos do padrão. Isso permite que o algoritmo saiba onde reiniciar a comparação quando uma discrepância é encontrada. Busca: O padrão é comparado com o texto. Quando uma discrepância é encontrada, a tabela de falhas é utilizada para determinar a próxima posição do padrão a ser comparada, evitando comparações redundantes. A complexidade temporal do KMP é $O(n + m)$, onde n é o comprimento do texto e m é o comprimento do padrão. Essa eficiência é alcançada porque o algoritmo nunca retrocede no texto; em vez disso, ele usa a informação da tabela de falhas para fazer saltos inteligentes no padrão.

Em resumo, o KMP é mais eficiente que o algoritmo de força bruta, especialmente para textos longos e padrões grandes, devido à sua capacidade de evitar comparações desnecessárias, tornando-o uma escolha preferida para muitos problemas de busca de padrões em strings.

2.4 Horspool's Algorithm

Horspool's Algorithm é uma variante do algoritmo de Boyer-Moore para busca de padrões em strings. Desenvolvido por Nigel Horspool em 1980, ele é conhecido por sua simplicidade e eficiência prática. O algoritmo pré-processa o padrão a ser buscado para criar uma tabela de deslocamentos, que indica quantas posições o padrão pode ser movido quando um caractere não corresponde. Durante a busca, o algoritmo se move pelo texto de forma a evitar comparações redundantes, resultando em uma eficiência média de $O(n/m)$, onde n é o tamanho do texto e m é o tamanho do padrão.

2.5 Boyer-Moore Algorithm

Boyer-Moore Algorithm é um algoritmo de busca de padrões em strings, desenvolvido por Robert S. Boyer e J Strother Moore em 1977. Este algoritmo é altamente eficiente, utilizando duas heurísticas principais: a heurística de bad-character e a heurística de good-suffix. A heurística de bad-character permite que o algoritmo pule múltiplas posições quando um caractere do texto não corresponde ao caractere do padrão, enquanto a heurística de good-suffix permite que o algoritmo mova o padrão de acordo com o sufixo correspondente mais à direita no padrão. Com uma complexidade de tempo no pior caso de $O(nm)$, ele é geralmente mais eficiente na prática, muitas vezes alcançando uma eficiência média de $O(n/m)$.

2.6 Hashing

Hashing é uma técnica usada para mapear dados de tamanho arbitrário a valores fixos, geralmente usados para criar tabelas de hash. Os valores mapeados, ou chaves, são armazenados em uma tabela de hash, permitindo buscas, inserções e exclusões rápidas.

Open Hashing (Separate Chaining)

Open Hashing, também conhecido como Separate Chaining, é uma técnica onde cada posição da tabela de hash aponta para uma lista vinculada de entradas que compartilham o mesmo valor de hash. Foi proposta como uma forma de lidar com colisões, onde múltiplas entradas têm o mesmo valor de hash. Cada entrada na lista vinculada contém uma chave e um valor, e o tempo de busca, inserção e exclusão depende do comprimento da lista vinculada, sendo $O(1)$ em média e $O(n)$ no pior caso, onde n é o número de entradas que compartilham o mesmo valor de hash.

Closed Hashing (Open Addressing) Closed Hashing, ou Open Addressing, é uma técnica onde todas as entradas são armazenadas na própria tabela de hash. Quando ocorre uma colisão, o algoritmo procura por outra posição na tabela de hash usando técnicas como linear probing, quadratic probing ou double hashing. Essa abordagem elimina a necessidade de estruturas adicionais, como listas vinculadas, e tem complexidade de tempo média para busca, inserção e exclusão de $O(1)$, embora o desempenho possa degradar para $O(n)$ no pior caso se muitas colisões ocorrerem.

2.7 B-Trees

B-Trees são estruturas de dados auto-balanceáveis usadas principalmente em sistemas de banco de dados e sistemas de arquivos para permitir acesso eficiente a dados armazenados em disco. Introduzidas por Rudolf Bayer e Edward M. McCreight em 1972, as B-Trees mantêm os dados ordenados e permitem buscas, inserções, exclusões e travessias sequenciais em tempo logarítmico. Uma B-Tree de ordem m é uma árvore balanceada onde cada nó pode ter no máximo m filhos e contém entre $\lceil m/2 \rceil$ e $m - 1$ chaves. A altura da árvore é mantida pequena, permitindo acesso rápido aos dados mesmo em grandes volumes de armazenamento.

3 METODOLOGIA

Para a realização de comparação dos algoritmos foi determinado uma linguagem padrão para a implementação de todos eles, de forma que todos os algoritmos fossem executados em ambientes similares para evitar imprecisões nos dados coletados.

3.1 Descrição do Hardware

Os testes foram realizados em apenas um computador de especificado logo abaixo:

3.1.1 Notebook Lenovo Ideapad Gaming 3i

- Processador: 11th Gen Intel(R) Core(TM) i5-11300H 3.11 GHz
- Memória RAM: 24GB DDR4 (1x8GB, 1x16GB)

3.2 Descrição do Software

Em relação ao software utilizado, foi utilizado o Sistema Operacional Windows 11 e a versão do Python 3.12.3, sendo que com uma peculiaridade, eles foram implementados utilizando Manjaro Linux, de versão estável, no WSL2. Como descrito mais detalhadamente:

3.2.1 Notebook Lenovo Ideapad Gaming 3i

- Sistema Operacional: Gentoo Linux 23.0 (WSL2) no Windows 11 Pro 23h2
- Versão do Python: 3.12.3

3.3 Implementação dos Algoritmos

Os algoritmos foram implementados utilizando a linguagem de programação Python. A escolha do Python se deve à sua simplicidade e ao seu amplo uso em ambientes acadêmicos e de pesquisa. Cada algoritmo foi encapsulado em uma função que recebe como entrada uma lista de números e retorna uma tupla contendo a lista ordenada, o número de comparações realizadas e o número de trocas efetuadas. Essa abordagem permite uma avaliação detalhada do desempenho de cada algoritmo, tanto em termos de tempo de execução quanto em operações realizadas.

3.4 Medição do Tempo de Execução

Para medir o tempo de execução dos algoritmos foi utilizado a função time da biblioteca padrão do Python para medir o tempo de execução dos algoritmos em milissegundos. Esse método é portátil e funciona em qualquer sistema operacional. A função é chamada antes do algoritmo de ordenação ser iniciado e após o algoritmo ser finalizado.

3.5 Número de Comparações e Trocas

Em cada função que encapsula os algoritmos, foram adicionadas duas variáveis do tipo inteiro para a realização da contagem do número de comparações e trocas, que foram incrementadas nos pontos apropriados do código.

3.6 Automatização dos testes

Para a automatização dos testes, foi criado um programa que itera a cada algoritmo uma série testes com parâmetros diferentes. Foram utilizados três tipos de listas: ordenada, inversa e aleatório. No qual cada tipo foi executado em listas com mil, dez mil, cinquenta mil e cem mil elementos. Como cada algoritmo retorna o número de comparações e trocas, e o programa retorna o tempo em milissegundos, esses dados foram adicionados em um arquivo para análise posterior.

3.7 Uso de Memória

Os algoritmos variam no uso de memória, conforme descrito abaixo:

- **Algoritmo de Força Bruta:** Não requerem espaço adicional significativo além da lista original.
- **Algoritmo KMP, Horspool:** Requer espaço adicional para as sublistas, o que pode ser um fator limitante em sistemas com memória restrita.
- **Algoritmo Buyer-Moore:** Requer espaço de memória suficiente para a execução.

4 CONCLUSÃO

Com este trabalho deu para avaliar bem o desempenho dos seis algoritmos de ordenação principais em termos de tempo de execução, número de comparações e trocas. Os resultados confirmam que os algoritmos com complexidade O(n log n) (Merge Sort, Quick Sort e Heap Sort) são significativamente mais eficientes para listas grandes do que os algoritmos de complexidade $O(n^2)$.

Recomendações

- **Força Bruta**: Adequados apenas para listas muito pequenas ou fins educacionais.
- **Insertion Sort**: Útil para listas pequenas ou quase ordenadas.
- **KMP, Buyer-Moore e Hosrpool**: Recomendado quando a estabilidade é crucial e há memória suficiente.
- **Buyer-Moore**: Geralmente a melhor escolha para desempenho rápido em listas grandes, mas cuidado com a quantidade de espaço requerido.

REFERÊNCIAS

Anany Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd edition, Pearson, 2011.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10-16, 1962.

Sultanullah Jadoon, Salman Faiz Solehria, Mubashir Qayum. Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study. *Department of Information Technology, Hazara University, Haripur Campus, 2. Sarhad University, Peshawar, 3. National University of Computer and Emerging Sciences, Peshawar Campus*. 115002-3838 IJECS-IJENS © April 2011 IJENS.

Owen Astrachan. *Bubble Sort: An Archaeological Algorithmic Analysis*. Computer Science Department, Duke University, December 9, 2003.

Rohit Joshi, Govind Singh Panwar, Preeti Pathak. *Analysis of Non-Comparison Based Sorting Algorithms: A Review*. Dept. of CSE, GEHU, Dehradun, India. International Journal of Emerging Research in Management & Technology, ISSN: 2278-9359, Volume 2, Issue 12, December 2013.

Paul Biggar, David Gregg. *Sorting in the Presence of Branch Prediction and Caches: Fast Sorting on Modern Computers*. Technical Report TCD-CS-2005-57, Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland, August 2005.

Ramesh Chand Pandey. *Study and Comparison of Various Sorting Algorithms*. Master's thesis, Thapar University, Patiala, July 2008.

Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. *A Comparison of Sorting Algorithms for the Connection Machine CM-2*. Carnegie Mellon University, MIT, NEC Research Institute, University of Texas, and Thinking Machines Corp, 2000.

Adarsh Kumar Verma and Prashant Kumar, *A New Approach for Sorting List to Reduce Execution Time*. Department of Computer Science and Engineering, Galgotias College of Engineering and Technology, Greater Noida, India. Octuber 2013.