CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA CELSO SUCKOW DA FONSECA – CEFET/RJ

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO - PPCIC

TEMA: REDUÇÃO DE BANDA DE MATRIZES

Eduardo Clinio e Valdinei Oliveira
Mestrandos em Ciência da Computação

Análise e Projeto de Algoritmos
Prof. Diego Nunes Brandão

Resumo: Neste trabalho é apresentado uma discussão para comparação dos métodos de soluções dos problemas de minimização de largura de Banda de Matrizes, através de algoritmos de computação. Os métodos são avaliados através dos custos, ou seja, o tempo gasto em segundos para a minimização das bandas. Através da linguagem Python, foi realizado diversos experimentos computacionais para comparar os resultados obtidos pelos métodos: Força Bruta, Guloso, Uma Proposição Heurística, Cuthill-McKee e Cuthill-McKee Reverso.

Palavras Claves: Banda de Matrizes, algoritmos, custos de computação.

Abstract: In this work, a discussion is presented to compare methods for solving Bandwidth Minimization Problems in Matrices using computational algorithms. The methods are evaluated based on costs, i.e., the time spent in seconds for bandwidth minimization. Using the Python language, various computational experiments were conducted to compare the results obtained by the following methods: Brute Force, Greedy, Heuristic Proposition, Cuthill-McKee, and Reverse Cuthill-McKee.

Keywords: Matrix Bandwidth, algorithms, computational costs.

1. INTRODUÇÃO

O problema de redução de banda de matrizes é crucial em muitas aplicações computacionais, onde a eficiência do armazenamento e a redução do tempo computacional são fundamentais. Uma matriz esparsa é uma matriz em que a maioria dos elementos possui o valor padrão (geralmente zero em contextos numéricos). Essas matrizes são comuns em problemas nos quais muitos elementos são zero, e armazenar todos esses elementos seria ineficiente em termos de espaço e tempo de processamento. Entre as definições, a proposta por (Carvalho et al., 2009), descrevem o problema a partir de uma matriz M quadrada e esparsa, podendo esta ser simétrica ou não. A banda de uma matriz é a banda diagonal que compreende as diagonais que contenham elementos não nulos mais distantes da diagonal principal em ambas as direções. O objetivo é, portanto, minimizar as distâncias das diagonais mais afastadas da diagonal principal da matriz. O lado esquerdo da diagonal principal denomina-se meia banda esquerda e ao lado direito da diagonal principal denomina-se meia banda direita. Segundo (Carvalho et al., 2009) o sistema algébrico linear Ax = b, onde A é uma matriz simétrica, esparsa e definida positiva de dimensão N×N. Métodos diretos como este se enquadram em duas classes básicas: os métodos esparsos gerais, que exploram todos os zeros em L, e os métodos de banda ou envelope, que exploram apenas aqueles zeros fora de um conjunto específico de posições de matriz em L. A possível vantagem de resolver o sistema permutado é que a solução pode exigir um número menor de operações aritméticas e/ou locais de armazenamento.

2. PROVA NP-COMPLETUDE

Segundo Garey e Johnson, 1979, para provar que um problema é NP-completo, normalmente seguem os seguintes passos: Mostrar que o problema está em NP, escolher um problema NP-completo conhecido, realizar uma redução polinomial e concluir a NP-completude. O problema de redução de banda de matriz não é um problema clássico da teoria da complexidade computacional, conforme discutido em sala de aula, (Brandão, 2023), o Problema do Caixeiro Viajante (TSP), o problema da mochila ou o problema da satisfação booleana (SAT). Portanto, realizar uma redução direta de um desses problemas clássicos para o problema de redução de banda de matriz pode não ser possível ou pode ser uma tarefa complexa. Como proposta, podemos usar um problema clássico chamado 3-SAT (um caso especial do Problema da Satisfação Booleana) para demonstrar a NP-completude do problema de redução de banda de matriz. O Problema 3-SAT é caraterizado por dado um conjunto de cláusulas booleanas, cada uma contendo exatamente três literais (variáveis booleanas ou suas negações), o problema 3-SAT pergunta se existe uma atribuição de valores verdadeiro/falso para as variáveis que satisfaça todas as cláusulas simultaneamente.

Para mostrar que o problema está em NP: Dada uma atribuição de valores verdadeiro/falso, podemos verificar em tempo polinomial se ela satisfaz todas as cláusulas.

Podemos usar o problema 3-SAT: com um problema clássico NP-completo. **A redução polinomial seria dada.** Dada uma instância de 3-SAT com variáveis $x1$, $x2$,…,xn e cláusulas $C1$,$C2$,…,Cm, construímos uma matriz A da seguinte maneira:

1. Cada linha da matriz A corresponde a uma cláusula Ci.
2. Cada coluna corresponde a uma variável xj ou sua negação $\neg xj$.
3. Se uma variável xj ou sua negação $\neg xj$ aparece na cláusula Ci, então Aij=1
4. Todas as outras entradas de A são zero.

Concluímos a NP-completude: A redução polinomial de 3-SAT para o problema de redução de banda de matriz garante que se pudermos resolver eficientemente o problema de redução de banda de matriz, poderíamos resolver eficientemente 3-SAT (Balbach, 2023). Como 3-SAT é NP-completo, concluímos que o problema de redução de banda de matriz também é NP-completo.

3. FORÇA BRUTA

A força bruta é uma técnica de resolução de problemas que explora todas as possíveis soluções, sem considerar estratégias mais inteligentes (Hristakeva e Shrestha, 2005). O algoritmo seleciona uma opção de uma maneira mais simples, direta ou óbvia. Ele repete essa tentativa até encontrar o resultado esperado. Principais caraterísticas: Avalia todas as opções disponíveis, geralmente, é computacionalmente intensiva, garante encontrar a solução ótima (se existir). Pode ser ineficiente para problemas complexos devido ao grande número de soluções a serem testadas.

3.1. Resultados Força bruta

Tam. Matriz	Tempo(Força Bruta)
1000	0.243797
2000	0.794105
3000	1.343107
4000	2.634516
5000	3.406565
6000	4.830639
7000	7.271306
8000	9.631424
9000	11.339759
10000	13.621956
11000	16.679887
12000	21.645370
13000	22.264120
14000	25.818868
15000	29.253847
16000	35.199701
17000	39.978010
18000	46.254095
19000	53.505479
20000	62.195560

Figura 1: . *Matrizes quadradas e esparsas com instâncias variando de 1.000 a 20.000 em intervalos de 1.000. Tempo em segundos*

4. GULOSO

O algoritmo guloso é uma abordagem de resolução de problemas em que, em cada passo, o algoritmo faz a escolha que parece ser a melhor no momento, com a expectativa de que essa escolha leve a uma solução global otimizada. A característica fundamental do algoritmo guloso é a tomada de decisões locais imediatamente benéficas, sem considerar implicações futuras. O termo ”guloso” refere-se à ideia de que o algo-

ritmo faz escolhas ”gananciosas”, optando pelo melhor caminho disponível em cada passo, sem se preocupar com o resultado final (Cormen et al., 2002). A principal característica do algoritmo guloso: escolhas locais ótimas, não volta atrás e não garante solução ótima global. É importante notar que, embora os algoritmos gulosos sejam eficientes e simples, nem sempre produzem soluções ótimas para todos os problemas. A escolha gulosa pode levar a soluções subótimas em alguns casos, e a validade do algoritmo depende da natureza específica do problema em questão.

4.1. Resultados Guloso

Tamanho da Matriz	Tempo Gasto (Guloso)
100	7.934230
110	9.951677
120	12.454042
130	14.932182
140	18.234554
150	21.203526
160	25.883170
170	28.967700
180	33.814210
190	38.528729
200	45.742847

Figura 2: . *Matrizes quadradas e esparsas com instâncias variando de 100 a 200 em intervalos de 10. Tempo em segundos.*

5. PROPOSIÇÃO HEURÍSTICA

Heurísticas são estratégias ou métodos aproximados que buscam encontrar soluções satisfatórias para problemas complexos, sem garantir a otimalidade. Características: Busca soluções de forma mais eficiente, usando regras práticas, que pode não garantir a melhor solução global. Mas visa encontrar uma solução aceitável em tempo razoável, e adaptar-se bem a problemas de grande escala ou com espaço de solução extenso. Uma proposta seria o código normalizar os números da matriz original entre 0 e 1, e gerar todas as permutações possíveis dos índices, além de escolher a permutação que minimiza os somatórios das linhas. A matriz resultante é então rearranjada de acordo com a melhor permutação.

5.1. Resultados Proposição Heurística

	Tamanho da Matriz	Tempo Gasto (Heurística)
0	1000	0.300342
1	2000	0.732874
2	3000	1.448493
3	4000	2.183799
4	5000	3.359456
5	6000	4.964066
6	7000	6.210109
7	8000	8.144787
8	9000	10.822167
9	10000	13.480871
10	11000	16.111915
11	12000	19.365576
12	13000	22.352625
13	14000	25.204412
14	15000	30.092970
15	16000	37.967810
16	17000	41.210262
17	18000	48.791801
18	19000	57.659708
19	20000	73.305214

Figura 3: . *Matrizes quadradas e esparsas com instâncias variando de 1.000 a 20.000 em intervalos de 1.000. Tempo em segundos.*

6. CUTHILL-McKEE e CUTHILL-McKEE REVERSO

As heurísticas de Cuthill-McKee e Cuthill-McKee reverso (Liu e Sherman, 1976) são técnicas usadas para reorganizar a ordem das variáveis (ou incógnitas) em sistemas de equações lineares, especialmente quando a matriz de coeficientes é esparsa e simétrica. Essas técnicas buscam reduzir a largura de banda da matriz, o que pode resultar em economia de armazenamento e acelerar algoritmos que operam sobre a matriz. **Cuthill-McKee (CMK):** O algoritmo Cuthill-McKee é uma heurística para reorganizar os índices das variáveis (ou nós) em um grafo representado pela matriz de coeficientes de um sistema de equações lineares. O objetivo é minimizar a largura de banda da matriz, facilitando a decomposição e outras operações eficientes. O algoritmo percorre o grafo em largura, reorganizando os nós de acordo com a ordem de visita. A reordenação resultante muitas vezes leva a uma matriz com uma largura de banda menor. **Cuthill-McKee Reverso (CMK Reverso)**: O CMK reverso é uma variação da heurística original. Em vez de seguir a ordem de visita original, essa técnica inverte essa ordem. O intuito é que, para certos tipos de matrizes esparsas, a inversão da ordem de visita pode levar a uma matriz com uma largura de banda ainda menor. Esta heurística foi observada empiricamente para ser eficaz em certos casos em que a reversão da ordem

de visita resulta em melhorias significativas em termos de armazenamento e eficiência de algoritmos de fatoração. Em geral, tanto o Cuthill-McKee quanto o Cuthill-McKee reverso são técnicas heurísticas, o que significa que não há garantia de encontrar a solução ótima, mas são projetadas para encontrar soluções boas o suficiente em tempo razoável. Essas heurísticas são frequentemente usadas em problemas de álgebra linear numérica e análise de grafos associada a sistemas físicos.

6.1. Resultados CUTHILL-McKEE e CUTHILL-McKEE REVERSO

Tam. Matriz	Tempo(Cuthill-McKee)	Tempo(Cuthill-McKee Reverso)
4000	0.109960	0.429072
8000	0.000000	1.792260
12000	0.015619	3.896586
16000	0.021400	8.782393
20000	0.031273	15.551048
24000	0.062690	28.816832
28000	0.130504	49.446158
32000	0.122569	106.258461
36000	0.110106	1292.167451
40000	0.135207	2263.546423

Figura 4: . *Matrizes quadradas e esparsas com instâncias variando de 4.000 a 40.000 em intervalos de 4.000. Tempo em segundos.*

6.2. Resultados CUTHILL-McKEE

Tam. Matriz	Tempo(Cuthill-McKee)	Tempo Acumulado
10000000	7.241643	7.241643
20000000	18.262763	25.504406
30000000	33.788611	59.293017
40000000	42.515649	101.808666
50000000	53.535936	155.344602
60000000	75.227901	230.572503
70000000	94.274439	324.846941
80000000	137.449201	462.296142
90000000	146.307438	608.603580
100000000	258.368767	866.972347

Figura 5: *Matrizes quadradas e esparsas com instâncias variando de 10.000.000 a 100.000.000 em intervalos de 10.000.000. Tempo em segundos.*

7. CONCLUSÕES

Neste estudo, notou-se que, dentre todos os experimentos realizados, o algoritmo de CUTHILL-McKEE (CMK) demonstrou, em termos tanto de armazenamento quanto de computação, uma superioridade de desempenho em comparação as técnicas de Força Bruta, Guloso e a Heurística Proposta. Utilizando como parâmetro o custo computacional de 20 minutos como estouro de memória, ficou evidente que o algoritmo de organização CMK supera a técnica CMK Reverso e os demais com grande eficiência computacional para a redução de banda de matrizes. De maneira intuitiva, podemos associar isso ao fato de que CMK procura minimizar a distância de ordenação entre um nó do grafo e seus vizinhos não ordenados, ao passo que CMK Reverso busca minimizar a distância entre um vértice e seus vizinhos ordenados.

REFERÊNCIAS

Balbach, F. J. (2023). The Cook-Levin theorem. *Archive of Formal Proofs*.

Brandão, D. N. (2023). Notas de Aula - Disciplina: Análise e Projeto de Algoritmos - Paradigmas de Projetos.

Carvalho, M. A. M., Junqueira, N. M. P., & Soma, N. Y. (2009). Uma Heurística para o problema de minimização de largura de banda em matrizes.

Cormen, T. H., Lee, C., Lin, E., Leiserson, C. E., Rivest, R. L., & Stein, C. (2002). Instructor's Manual.

Garey, M. R., & Johnson, D. S. (1979). Computers and intractability. *A Guide to the*.

Hristakeva, M., & Shrestha, D. (2005). Different approaches to solve the 0/1 knapsack problem. *The Midwest Instruction and Computing Symposium*.

Liu, W.-H., & Sherman, A. H. (1976). Comparative analysis of the Cuthill–McKee and the reverse Cuthill–McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, *13*(2), 198–213.

8. ANEXOS

Força Bruta

```
In [ ]: import numpy as np
from scipy.sparse import csr_matrix
import time
import pandas as pd
from scipy.sparse.csgraph import reverse_cuthill_mckee

def generate_sparse_matrix(size):
    row = np.random.randint(0, size, size * 2)
    col = np.random.randint(0, size, size * 2)
    data = np.random.randint(1, 10, size * 2)  # Números inteiros entre 1 e 10
    return csr_matrix((data, (row, col)), shape=(size, size))

def normalize_and_reduce(matrix):
    size = matrix.shape[0]
    best_permutation = None
    best_row_sums = None

def bandwidth_reduction(matrix):
    sparse_matrix = csr_matrix(matrix)
    permutation = reverse_cuthill_mckee(sparse_matrix, symmetric_mode=True)
    reduced_matrix = matrix[:, permutation]
    return reduced_matrix

# Lista para armazenar os DataFrames de cada iteração
dfs = []

# Loop de tamanhos de matriz de 1.000 a 20.000, de 1.000 em 1.000
for size in range(1000, 20001, 1000):
    sparse_matrix = generate_sparse_matrix(size)

    Mede o tempo de execução para a abordagem de força bruta
start_time_force_bruta = time.time()
_, _ = normalize_and_reduce(sparse_matrix)
elapsed_time_force_bruta = time.time() - start_time_force_bruta

        # Cria um DataFrame com os resultados
    df = pd.DataFrame({
        'Tam. Matriz': [size],
        'Tempo(Força Bruta)': [elapsed_time_force_bruta],

    dfs.append(df)

# Concatena todos os DataFrames em um único DataFrame
results_df = pd.concat(dfs, ignore_index=True)

# Imprime os resultados em uma única linha
print(results_df.to_string(index=False))
```

Figura 6: *Código Algoritmo Força Bruta*

Proposta de Heurística

```
In [24]: import numpy as np
from scipy.sparse import csr_matrix
import time
import pandas as pd

def generate_sparse_matrix(size):
    row = np.random.randint(0, size, size * 2)
    col = np.random.randint(0, size, size * 2)
    data = np.random.randint(1, 10, size * 2)  # Números inteiros entre 1 e 10
    return csr_matrix((data, (row, col)), shape=(size, size))

def normalize_and_reduce(matrix):
    size = matrix.shape[0]
    best_permutation = None
    best_row_sums = None

    # Normaliza os números da matriz entre 0 e 1
    normalized_matrix = matrix / np.max(matrix)

    # Se a matriz for unidimensional, não é necessário fazer permutações
    if normalized_matrix.ndim == 1:
        best_permutation = np.arange(size)
    else:
        # Gera todas as permutações possíveis dos índices
        for permutation in np.argsort(normalized_matrix.toarray().T, axis=1):
            # Aplica a permutação à matriz
            permuted_matrix = normalized_matrix[:, permutation]

            # Calcula os somatórios das linhas
            row_sums = np.sum(permuted_matrix, axis=1)

            # Atualiza a melhor permutação se encontrarmos uma com somatórios menores que 0.5
            if best_permutation is None or np.sum(row_sums < 0.5) < np.sum(best_row_sums < 0.5):
                best_permutation = permutation
                best_row_sums = row_sums

    # Aplica a melhor permutação à matriz original
    reduced_matrix = matrix[:, best_permutation]

    # Aplica a melhor permutação à matriz original
    reduced_matrix = matrix[:, best_permutation]

    return best_permutation, reduced_matrix

# Lista para armazenar os DataFrames de cada iteração
dfs = []

# Loop de tamanhos de matriz de 5 a 50, de 5 em 5
for size in range(1000, 20001, 1000):
    sparse_matrix = generate_sparse_matrix(size)

    # Mede o tempo de execução para a abordagem de força bruta
    start_time = time.time()
    best_permutation, reduced_matrix = normalize_and_reduce(sparse_matrix)
    elapsed_time = time.time() - start_time

    # Cria um DataFrame com os resultados
    df = pd.DataFrame({'Tamanho da Matriz': [size], 'Tempo Gasto (Heurística)': [elapsed_time]})
    dfs.append(df)

# Concatena todos os DataFrames em um único DataFrame
results_df = pd.concat(dfs, ignore_index=True)

# Imprime o DataFrame
print(results_df)
```

Figura 8: *Código Proposta de Heurística*

Guloso

```
In [28]: import numpy as np
from scipy.sparse import csr_matrix
import pandas as pd
import time

def generate_sparse_matrix(size):
    row = np.random.randint(0, size, size * 2)
    col = np.random.randint(0, size, size * 2)
    data = np.random.randint(1, 10, size * 2)  # Números inteiros entre 1 e 10
    return csr_matrix((data, (row, col)), shape=(size, size))

def bandwidth(matrix):
    # Calcula a largura de banda da matriz
    row_indices, col_indices = matrix.nonzero()
    bandwidth = np.max(np.abs(row_indices - col_indices))
    return bandwidth

def greedy_bandwidth_reduction(matrix):
    size = matrix.shape[0]
    permutation = np.arange(size)
    original_bandwidth = bandwidth(matrix)

    for i in range(size):
        best_bandwidth_reduction = 0
        best_position = i

        for j in range(i, size):
            # Tenta mover a linha/coluna j para a posição i e calcula a redução na largura de banda
            matrix[:, [i, j]] = matrix[:, [j, i]]
            matrix[[i, j], :] = matrix[[j, i], :]
            new_bandwidth = bandwidth(matrix)

            # Se a mudança reduzir a largura de banda, atualiza as variáveis
            if original_bandwidth - new_bandwidth > best_bandwidth_reduction:
                best_bandwidth_reduction = original_bandwidth - new_bandwidth
                best_position = j

            # Reverte a mudança para testar a próxima posição
            matrix[:, [i, j]] = matrix[:, [j, i]]
            matrix[[i, j], :] = matrix[[j, i], :]

        # Move a linha/coluna selecionada para a posição correta
        matrix[:, [i, best_position]] = matrix[:, [best_position, i]]
        matrix[[i, best_position], :] = matrix[[best_position, i], :]
        permutation[i], permutation[best_position] = permutation[best_position], permutation[i]
```

Figura 7: *Código Algoritmo Guloso*

Cuthill-McKee e Cuthill-McKee Reverso

[3]:
```python
import numpy as np
from scipy.sparse import csr_matrix
import time
import pandas as pd
from scipy.sparse.csgraph import reverse_cuthill_mckee

def generate_sparse_matrix(size):
    row = np.random.randint(0, size, size * 2)
    col = np.random.randint(0, size, size * 2)
    data = np.random.randint(1, 10, size * 2)  # Números inteiros entre 1 e 10
    return csr_matrix((data, (row, col)), shape=(size, size))

def normalize_and_reduce(matrix):
    size = matrix.shape[0]
    best_permutation = None
    best_row_sums = None

def bandwidth_reduction(matrix):
    sparse_matrix = csr_matrix(matrix)
    permutation = reverse_cuthill_mckee(sparse_matrix, symmetric_mode=True)
    reduced_matrix = matrix[:, permutation]
    return reduced_matrix

# Lista para armazenar os DataFrames de cada iteração
dfs = []

# Loop de tamanhos de matriz de 4.000 a 40.000, de 4.000 em 4.000
for size in range(4000, 40001, 4000):
    sparse_matrix = generate_sparse_matrix(size)

    # Mede o tempo de execução para Cuthill-McKee
    start_time_cmk = time.time()
    _ = bandwidth_reduction(sparse_matrix)
    elapsed_time_cmk = time.time() - start_time_cmk

    # Mede o tempo de execução para Cuthill-McKee Reverso
    start_time_rcm = time.time()
    _ = bandwidth_reduction(sparse_matrix.toarray())
    elapsed_time_rcm = time.time() - start_time_rcm

    # Cria um DataFrame com os resultados
    df = pd.DataFrame({
        'Tam. Matriz': [size],
       # 'Tempo(Força Bruta)': [elapsed_time_force_bruta],
        'Tempo(Cuthill-McKee)': [elapsed_time_cmk],
            'Tempo(Cuthill-McKee Reverso)': [elapsed_time_rcm]
    })
    dfs.append(df)

# Concatena todos os DataFrames em um único DataFrame
results_df = pd.concat(dfs, ignore_index=True)
```

Figura 9: *Código CUTHILL-McKEE e CUTHILL-McKEE Reverso*