MS529 - Fluxo em Redes
Projeto computacional

Bernardo Correia Prados Nascimento RA: 245837

Junho 2025

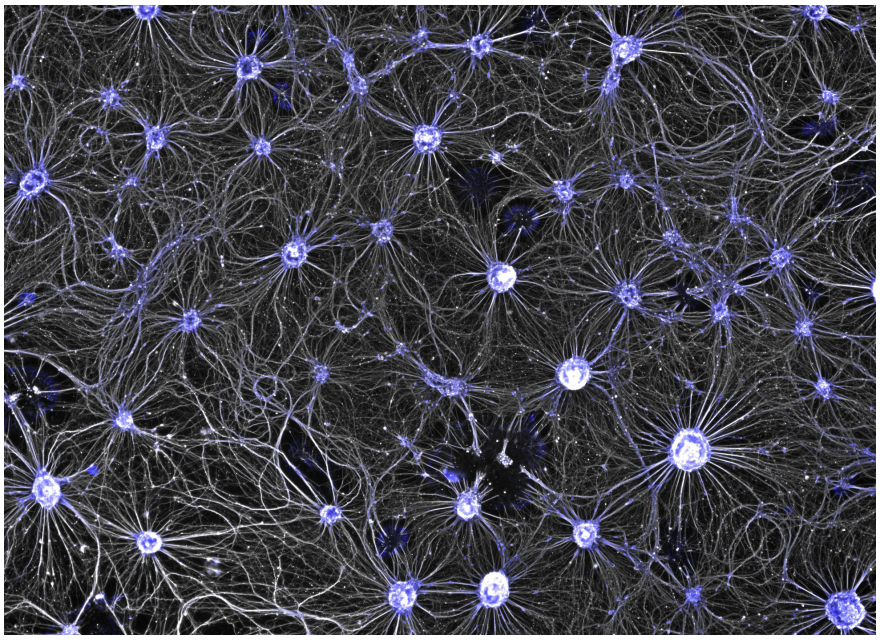

Imagem 1: Microfotografia de neurônios motores derivados de células-tronco pluripotentes induzidas em humanos, por Sarah Hörner.

1 Introdução

O presente projeto visa ao estudo e implementação de técnicas de otimização de armazenamento de dados, com base em algoritmos de construção de Árvores Geradoras Mínimas (AGM). O tema é brevemente apresentado na quarta seção do 13° capítulo *Minimum Spanning Trees* em *Network flows: theory, algorithms, and applications* [1]. Mais significativo foi o amparo teórico e técnico oriundo do artigo *Storage reduction through minimal spanning trees and spanning forests* [2].

A fim de contextualizar as áreas de aplicação dessas técnicas, supõe-se um cenário base onde é preciso armazenar múltiplos vetores (amostras, fichas de informação, sequências genéticas), cujas entradas variam dentro de um conjunto finito e enumerável de valores. Além disso, também assume-se que as discrepâncias entre as amostras, isto é, a diferença entre valores de entradas de mesma posição, ocorrem com baixa frequência.

Sob tais condições, o armazenamento direto dos vetores em uma estrutura de *array* bidimensional compreende estratégia intuitiva e, contudo, proporcionalmente ingênua. Uma solução alternativa consiste no armazenamento baseado em vetores de diferenças. Por essa ótica, usufrui-se da similaridade entre entidades de cada amostra, de modo a guardar um vetor referência e as diferenças relativas a ele nos demais vetores. A título de exemplo, suponha que se tem em mãos três vetores A, B, C. Ao invés de armazená-los sequencialmente em uma matriz [A, B, C], pode-se salvar o vetor B como referência, e então as posições e os valores das diferenças entre A e B no vetor dif_{BA}, e, por fim, posições e valores das diferenças entre B e C no vetor dif_{BC}. Dessa forma, resta guardar a matriz [B, dif_{BA}, dif_{BC}].

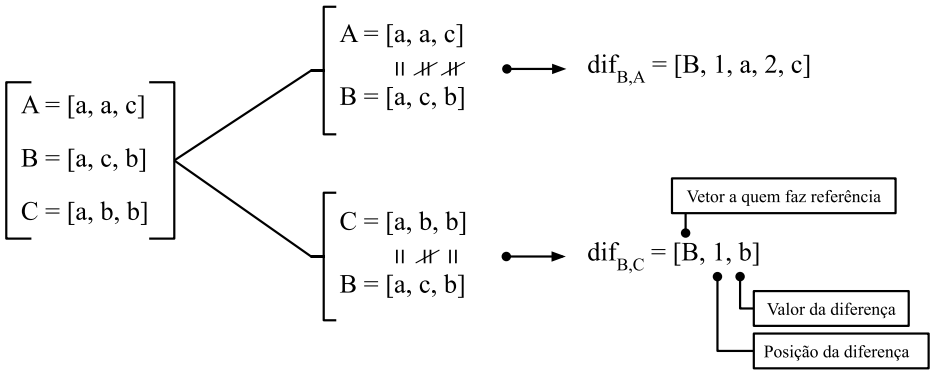

Imagem 2: Esquema ilustrativo do vetor de diferenças entre amostras.

Para conjuntos de dados de alto volume, e em consonância com as hipóteses supracitadas, espera-se que a técnica de armazenamento por similaridade reduza, consideravelmente, a memória utilizada. A fim de otimizar a geração das

matrizes de similaridade, lança-se mão de AGMs, de modo a encontrar os caminhos mínimos para alcançar um vetor (equivalente a um nó na árvore) partindo de um vetor referência (nó raiz). Por conseguinte, outro desafio intrínseco à abordagem em estudo consiste na escolha do vetor de referência. Complementarmente, mais um aspecto de grande relevância computacional são as variadas implementações, através de diferentes estruturas de dados, dos algoritmos de construção de AGMs.

2 Construção da rede de similaridade

Considere um conjunto de n dados, cada um sendo um vetor com m entradas. Ou seja, as i-ésima e j-ésima amostras são da forma

$$D_i = [x_1^i, x_2^i, \cdots, x_m^i]$$

e

$$D_j = [x_1^j, x_2^j, \cdots, x_m^j] \quad ,$$

em que $x_k \in U_k, \quad k = 1, 2, \cdots, m$, sendo U_k o conjunto dos possíveis valores assumidos por x_k.

Dessa maneira, define-se a função auxiliar f_k como indicadora de discrepância na k-ésima entrada entre dois vetores de dados distintos. Isto é, $f_k : U_k \times U_k \rightarrow \{0, 1\} \quad k = 1, 2, \cdots, m$, com

$$f_k(x_k^i, x_k^j) = \begin{cases} 0, & x_k^i = x_k^j \\ 1, & c.c. \end{cases} \quad .$$

Daí, pode-se definir a distância $d_{i,j}$ entre amostras i e j como a soma do número de discrepâncias. Formalmente, $d_{i,j} : (U_1 \times U_2 \times \cdots \times U_m)^2 \rightarrow \mathbb{N}, \quad i, j \in \{1, 2, \cdots, n\}$, com

$$d_{i,j} = \sum_{k=1}^{m} f_k(x_k^i, x_k^j) \quad .$$

A técnica de armazenamento tem início com o cálculo do número de discrepâncias entre todo par de vetores (i, j), a fim de mapear as distâncias relativas do conjunto de dados. Consequentemente, para um conjunto de n amostras, são calculadas $C_2^n = \frac{n(n-1)}{2}$ distâncias, em que C_2^n é o número de combinações tomadas 2 a 2 no conjunto.

Munidos das definições acima, pode-se construir a rede de similaridade como um grafo não direcionado $G = (N, A)$, com $|N| = n$, em que cada amostra representa um nó. Além disso, visto que cada nó i se relaciona aos demais nós $j \neq i$ a partir do valor $d_{i,j}$, tem-se que $|A| = \frac{n(n-1)}{2}$. O resultado - ilustrado a seguir por um exemplo com 15 vetores - consiste numa rede densa a qual indica o "custo de reconstrução" de um vetor i a partir de j.

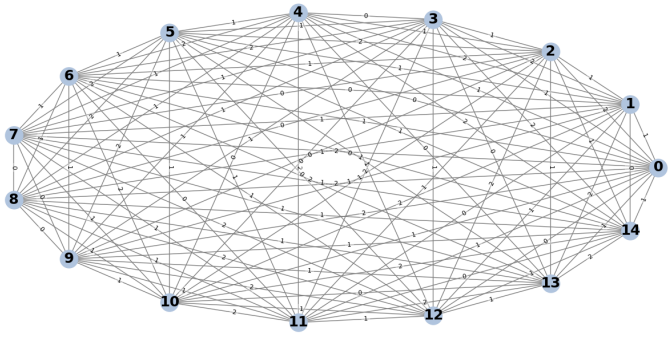

Imagem 3: Ilustração do grafo de similaridade de conjunto com 15 vetores.

3 Construção da árvore geradora mínima

Uma vez construído o grafo de similaridade, objetiva-se encontrar os caminhos de menor custo que interligam todos os nós. Sob a perspectiva da similaridade entre as amostras, é preciso delinear, provida uma amostra referência, o caminho mais curto (com menos discrepâncias) para a reconstrução de um vetor qualquer. Para tanto, voltam-se os esforços para a obtenção da AGM.

Nesse projeto, lançou-se mão, por fins comparativos, dos métodos de Prim e Kruskal para a construção das árvores, cujas implementações estão no arquivo `min_span_tree.py`. Ao longo da implementação, também tomou-se liberdade para explorar o perfil de comportamento dos algoritmos, em termos de tempo de execução e armazenamento computacional poupado, através de diferentes estruturas de dados. A implementação de estruturas seguiu os passos esclarecedores do professor Paulo Feofiloff do Departamento de Ciência da Computação do IME-USP, cujas notas de aula sobre os assuntos em muito contribuíram para a compreensão dos algoritmos [3].

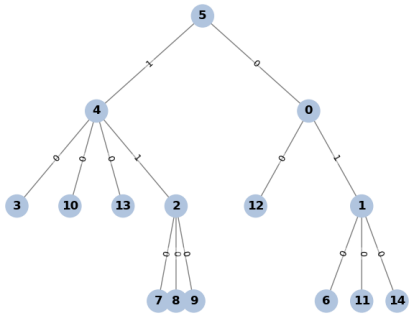

Imagem 4: Ilustração da AGM resultante de conjunto com 15 vetores.

3.1 Algoritmo de Kruskal

Restrito a grafos não direcionados, o algoritmo de Kruskal desponta como um dos principais candidatos para tratar a rede de similaridade obtida anteriormente. Em linhas gerais, ele se desenvolve assim: inicialmente, ordenam-se as arestas do grafo em ordem não decrescente de custo; insere-se cada uma delas à lista de arestas da AGM, à medida que ciclos não são formados; caso o sejam, descarta-se a aresta. O algoritmo encerra quando o número de arestas da árvore é igual ao número de nós do grafo a menos de uma unidade.

Sendo assim, em termos de complexidade algorítmica, duas fragilidades se destacam no método: em primeiro lugar, a necessidade de ordenação da lista de arestas; em segundo, a averiguação de formação de ciclos na árvore em construção. O primeiro ponto é otimizado via aplicação de *heapsort*, resultando, no caso de uma lista com $a = \frac{n(n-1)}{2}$ arestas, em uma complexidade de $O(a \log a) = O(\frac{n(n-1)}{2} \log \frac{n(n-1)}{2}) = O(n^2 \log n^2) = O(n^2 \log n)$.

No que tange ao segundo ponto, uma forma direta de sanar o sub-problema seria a varredura em profundidade da árvore, no mínimo uma vez a cada iteração do algoritmo. No entanto, visando a eficiência computacional, a fim de validar a formação de ciclos recorreu-se à estrutura de dados *disjoint-set*: ao definir um sistema de partição aos nós do grafo, ela possibilita a verificação rápida quanto à partição de dois elementos distintos - isto é, se dois nós compartilham a mesma partição, a adição de um arco entre eles gera, invariavelmente, um ciclo na árvore. Também conhecida como união-busca, a estrutura garante a checagem de partições em complexidade $O(1)$.

Por conseguinte, a implementação de Kruskal supracitada leva à complexidade final de $O(a \log a) + O(1) = O(n^2 \log n + 1) = O(n^2 \log n)$. Sob esse raciocínio, é fácil ver que o gargalo do algoritmo reside na etapa de inicialização, durante a ordenação dos arcos. A implementação da estrutura *union-find* é feita no arquivo `union_find.py`, enquanto a do algoritmo de *heapsort* se dá em `heap_graph.py`. Abaixo, o pseudoalgoritmo de Kruskal.

Algoritmo 1: KRUSKAL

Entrada: N, A, c
Saída: A_arv, c_arv

início
  $A_{ord} :=$ **ordena**(A)
  $A\_arv := \{\}$
  $c\_arv := 0$
  $contador = 0$
  enquanto $|A\_arv| \neq |N| - 1$ **faça**
    $aresta := A_{ord}(contador)$
    se *a inclusão de aresta em A_arv gera ciclo* **então**
      Não insira
    senão
      $A\_arv \leftarrow A\_arv \cup aresta$
      $c\_arv \leftarrow c\_arv + c(aresta)$
    fim
    $contador \leftarrow contador + 1$
  fim
fim

3.2 Algoritmo de Prim

Também voltado a grafos não direcionados, o algoritmo de Prim funciona da seguinte maneira: particiona-se o grafo em dois, uma parte contendo o nó de partida s, e outra contendo os demais nós; em seguida, procura-se o arco (s, j) de menor custo tal que j pertença à segunda partição; adiciona-se j à primeira partição e exclui-se da segunda. Repete-se o processo para os nós da primeira partição, até que a segunda se esvazie.

Uma vez descrito o algoritmo, é possível apontar como sua fragilidade a etapa de busca pelos arcos mínimos (i, j) os quais respeitem a condição de i pertencer à árvore em construção, e j ao outro corte. A resolução direta do subproblema incluiria, à custo polinomial, a varredura das arestas que conectam ambas partições do grafo. Contudo, mediante a implementação da estrutura *min-heap* binária (presente em `heap_graph.py`), rearranjou-se o conjunto de arestas em uma fila de prioridade de custo mínimo. De modo subsequente, a varredura completa das arestas foi substituída pela simples extração do elemento de maior prioridade da *heap* - seguida das eficientes operações de preservação da estrutura.

Dessa forma, dado um grafo com n nós e $\frac{n(n-1)}{2}$ arestas, a implementação de Prim por árvore binária rende uma complexidade de $O(\frac{n(n-1)}{2} \log n) = O(n^2 \log n)$ - a mesma ordem obtida previamente com Kruskal. Abaixo, descreve-se um pseudoalgoritmo para Prim.

Algoritmo 2: PRIM

Entrada: N, A, c, s
Saída: A_arv, c_arv

início
  $P := \{s\}$
  $P' := N \setminus \{s\}$
  $A\_arv := \{\}$
  $c\_arv := 0$
  enquanto $P' \neq \{\}$ **faça**
    determine i e j tal que $c(i,j) = \min_{i \in P, j \in P'} \{c(i,j) | (i,j) \in A\}$
    $P \leftarrow P \cup \{j\}$
    $P' \leftarrow P' \setminus \{j\}$
    $A\_arv \leftarrow A\_arv \cup (i,j)$
    $c\_arv \leftarrow c\_arv + c(i,j)$
  fim
fim

A imagem a seguir auxilia na visualização das ordens de complexidade de cada algoritmo. Nos experimentos que ela retrata, foram gerados conjuntos aleatórios de 100, 150, 200, $\cdots$, 950 vetores, cada um com 10 variáveis. Com a mesma probabilidade, as entradas dos vetores assumiram os valores 0 ou 1. Em seguida, construíram-se as redes de similaridade, e por fim aplicaram-se: Prim com varredura das arestas (em verde); Prim com *min-heap* (em azul); e Kruskal (em laranja).

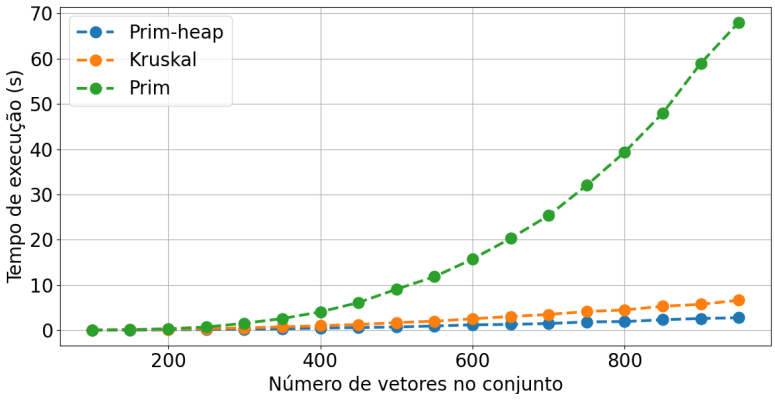

Imagem 5: Relação entre o tempo de execução e o número de vetores no conjunto de amostras para diferentes métodos

A partir do gráfico, evidencia-se a significativa discrepância, em tempo de execução, entre a abordagem direta do método de Prim, e a implementação por

fila de prioridade. Ademais, destaca-se a proporcionalidade entre Prim por *min-heap* e Kruskal - corroborando os cálculos anteriores. Em suma, os resultados jogam luz à importância de adequar os algoritmos selecionados a estruturas de dados capazes de otimizar, senão todo o processo, etapas pontuais e, sobretudo, inexoráveis gargalos.

3.3 Raízes e direcionamento

Uma vez construída a AGM para a rede de similaridade, ainda resta o problema de definição da amostra referência para o conjunto. Na árvore, tal vetor compreenderia o nó raiz, de onde emergem os arcos de diferenças para reconstrução dos demais nós. Por fim, deve-se atribuir, partindo do nó raiz, direção aos arcos da árvore - a fim de direcionar um plano de alterações para a obtenção de qualquer amostra do conjunto de dados.

Idealmente, o vetor referência deveria ser aquele de maior similaridade com o resto do conjunto - isto é, o dado i que minimiza $\sum_{j \in N} d_{i,j}$. Sob tal perspectiva, intui-se que o melhor candidato para a raiz da AGM compreende o nó cuja soma dos custos para alcançar qualquer outro nó é mínima. No artigo de Kang, Lee, Chang e Chang, é provado que essa propriedade é satisfeita pelo centroide da árvore.

No mesmo artigo, define-se o centroide da árvore como segue: "Seja T uma árvore com n pontos. E sejam também $Q_1, Q_2, \cdots, Q_m$ pontos adjacentes a um ponto P qualquer em T. Para cada i, $1 \leq i \leq m$, seja T_i (com $Q_i \in T_i$) a subárvore de n_i pontos obtida ao retirar o arco (P, Q_i) de T. Diz-se que P é centroide da árvore T se, e somente se, $n_i \leq n/2$ para todo i tal que $1 \leq i \leq m$".

Com base na definição supracitada, e no pseudoalgoritmo breve e ulteriormente sugerido no artigo, foi possível formular um código recursivo para determinar o nó centroide de uma árvore. Em linhas gerais, o algoritmo pode ser descrito da seguinte maneira: dado um nó, são varridos todos os nós a quem é ligado; para cada nó filho encontrado, a busca por descendentes é recursivamente chamada; cada chamada da função finaliza apenas quando se alcança um nó que possui apenas ligação com seu antecessor, então seu número de descendentes é igual a 1; o nó original tem (número de descendentes) = (número de descendentes de seus filhos) + 1, logo a cada "volta" das chamadas recursivas, atualiza-se o número de descendentes do nó atual como (número de descendentes de seus filhos) + 1. Ao longo da busca, mantém-se controle sobre os nós para checar se o centroide - primeiro nó o qual atinge um número de descendentes maior ou igual à metade do número de nós da árvore - já foi encontrado.

Algoritmo 3: Busca por descendentes

Entrada: N, A, nó atual, centroide, pred, descend
Saída: centroide (nó centroide ou `None`)

```
1  início
2      se todos os vizinhos de nó atual já foram visitados então
3          descend(nó atual) ← descend(nó atual) + 1
4      senão
5          para todo nó vizinho ao nó atual faça
6              pred(nó vizinho) = nó atual
7              centroide = Busca por descendentes
8              descend(nó atual) ← descend(nó atual) + descend(nó
                vizinho)
9              se descend(nó atual) ≥ |N|/2 e ainda não se encontrou o
                centroide então
10                 centroide = nó atual
11             fim
12         fim
13     fim
14     retorna centroide
15 fim
```

Acima, descreve-se mais formalmente o pseudoalgoritmo da busca por descendentes. A princípio, a chamada inicial da função é feita com os parâmetros: pred (lista de predecessores de cada nó) = lista de `None`; descend (lista de descendentes de cada nó) = lista de zeros; centroide (número do nó centroide) = `None`. A implementação computacional é feita, e documentada, no arquivo `label_node_search.py`.

A seguir, é exibido um exemplo do algoritmo supracitado, aplicado em uma árvore com 5 nós. Dessa forma, qualifica-se como centroide o nó que primeiro ultrapassar a marca dos $\lfloor \frac{|N|}{2} \rfloor = \lfloor \frac{5}{2} \rfloor = 2$ descendentes. Mais adiante, esse exemplo com 5 nós será reutilizado para indicar o direcionamento da árvore geradora mínima.

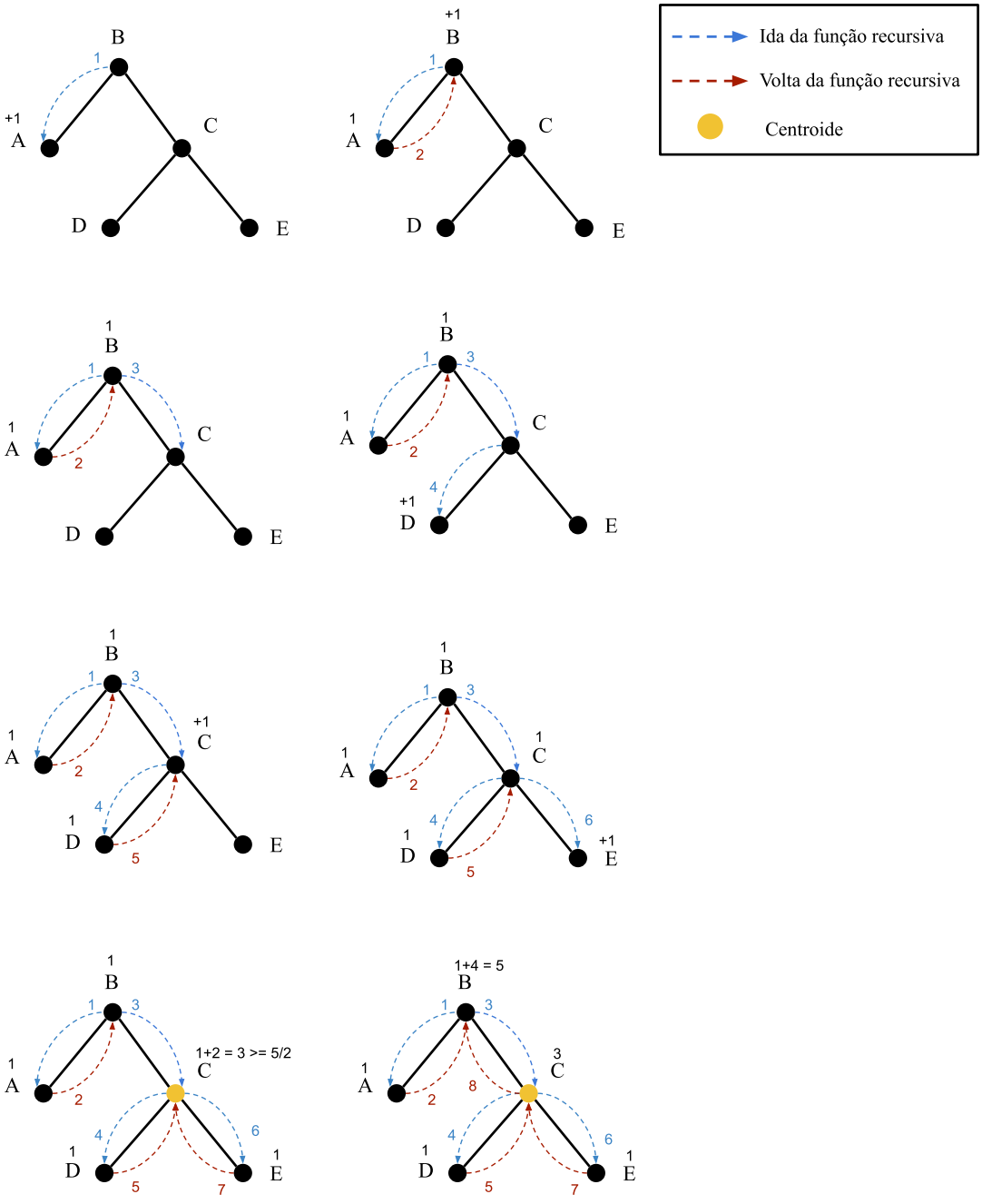

Imagem 6: Esquema ilustrativo do algoritmo recursivo para determinação do centroide da árvore, via enumeração do número de descendentes para cada nó. Acima dos nós, é contabilizado o número de descendentes descobertos até então. Acima das linhas pontilhadas, é indicada a ordem das idas/voltas recursivas.

Por fim, resta o direcionamento para fora da árvore, a partir da raiz. Para tanto, é possível acoplar, ao algoritmo dos Nós Marcados, a contínua adição de arestas direcionadas a uma nova lista de arcos. A função incumbida de tal papel também se encontra no arquivo `min_span_tree.py`. Para além do direcionamento dos arcos, também se usufrui da lista de predecessores diretos de cada nó. Sob o olhar da similaridade entre vetores, dada uma aresta direcionada (i, j), tem-se que o vetor j tem, dentre todos as amostras, mais semelhança com i.

Complementarmente, vale ressaltar que uma válida modificação nos algoritmos de construção das árvores geradoras mínimas, consiste em priorizar, ao longo da seleção de arestas de custo mínimo, ligações a vértices com maior número de vizinhos. Dessa maneira, prioriza-se a formação de árvores superficiais, ou seja, com altura menor, as quais providenciam caminhos mais curtos de reconstrução de amostras.

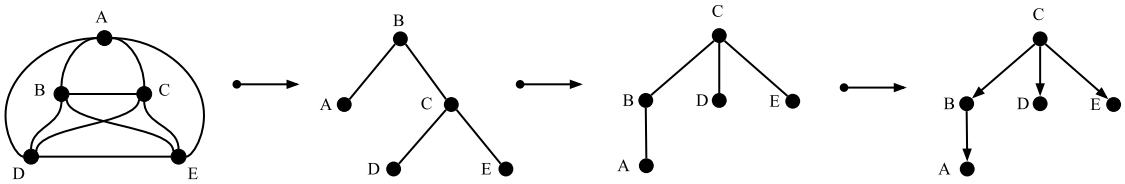

Imagem 7: Esquema ilustrativo da evolução da rede de similaridade, passando pela construção da AGM, e então para a definição do centroide, e o subsequente direcionamento para fora - nessa ordem.

4 Redução de armazenamento

As ferramentas e algoritmos anteriores abrem caminho direto para a construção da matriz de armazenamento ótimo. Uma vez performada a série de buscas sobre a rede densa de similaridade, deve-se prosseguir da seguinte maneira:

1. Armazena-se integralmente a raiz da AGM;
2. Para cada nó i diferente da raiz, armazena-se o número j do seu nó predecessor direto na árvore, além das diferenças relativas e suas posições;

Nos programas que acompanham o presente projeto, a implementação dessas funções - e outras referentes à reconstrução de amostras originais - se dá em `project.py`. Dessa maneira, supondo que cada entrada dos vetores ocupa uma única posição na memória, pode-se afirmar que a técnica exige o armazenamento final de: n apontadores para cada amostra (ou seja, 1 apontador para o vetor referência, e $n-1$ apontadores para os demais vetores de diferenças); m espaços para alocar o conteúdo do vetor referência; para cada vetor não-referência i, deve-se armazenar o vetor ao qual faz referência, além das d_i diferenças e suas d_i posições, então $1 + 2d_i$. Portanto, o armazenamento total S é dado por

$$S = n + m + \sum_{i=1}^{n-1}(1 + 2d_i) \quad .$$

Tal análise permite comparar o armazenamento resultante do método direto, com a técnica baseada em árvores geradoras mínimas. Para o armazenamento bidimensional, tem-se a alocação de $\overline{S} = m \cdot n$ espaços na memória. Logo, a técnica em estudo torna-se superior quando:

$$\begin{aligned} S < \overline{S} &\Leftrightarrow n + m + \sum_{i=1}^{n-1}(1 + 2d_i) < m \cdot n \\ &\Leftrightarrow n + m + n - 1 + 2\sum_{i=1}^{n-1} d_i < m \cdot n \\ &\Leftrightarrow 2n + m - 1 + 2\sum_{i=1}^{n-1} d_i < m \cdot n \\ &\Leftrightarrow 2\sum_{i=1}^{n-1} d_i < m \cdot n - m - 2n + 1 \quad , \end{aligned}$$

e, visto que $n >> 1$, pode-se dizer que $n + 1 \approx n + 2$, daí

$$\begin{aligned} S < \overline{S} &\Leftrightarrow 2\sum_{i=1}^{n-1} d_i < m \cdot n - m - 2n + 2 \\ &\Leftrightarrow 2\sum_{i=1}^{n-1} d_i < (m-2)(n-1) \\ &\Leftrightarrow \frac{1}{n-1}\sum_{i=1}^{n-1} d_i < \frac{1}{2}(m-2) \quad , \end{aligned}$$

Agora, note que a quantidade $\sum_{i=1}^{n-1} d_i$ equivale ao custo total da AGM. Assim, a fração do lado esquerdo representa o custo médio das arestas da árvore - uma vez que possui $n - 1$ arestas. Denominando o valor médio como $\hat{d}$, tem-se que:

$$S < \overline{S} \Leftrightarrow \hat{d} < \frac{1}{2}(m-2) \quad .$$

Apesar da condição suficiente e necessária para que o armazenamento resultante da técnica seja menor que o método tradicional, a checagem dessa condição ainda exige a construção da árvore geradora mínima direcionada, para então calcular o custo médio de suas arestas. Seria vantajoso a dedução de um minorante para o número de amostras, a partir do qual tem-se certeza que a técnica supera o armazenamento direto. Para tanto, parte-se do pior caso possível para o conjunto de dados. Suponha que o conjunto é formado por vetores de tamanho m

e que cada entrada pode assumir k valores distintos. Logo, existem k^m vetores diferentes. No pior dos cenários, o conjunto de dados possui exatamente os k^m vetores distintos. Para todo m natural, é possível construir indutivamente uma AGM não orientada cujo custo de todas as arestas é 1. A saber, ordenam-se os vetores de modo que: o primeiro deles será a amostra $[1, 1, \cdots, 1, 1]$, em seguida, $[1, 1, \cdots 1, 2]$, daí $[1, 1, \cdots 1, 3]$, e assim até $[1, 1, \cdots 1, k]$. O próximo nó da árvore corresponde ao vetor $[1, 1, \cdots 2, k]$, daí $[1, 1, \cdots 3, k]$, até $[1, 1, \cdots k, k]$. O processo é repetido até que se alcance o último nó $[k, k, \cdots k, k]$. Assim, constrói-se a árvore de custo médio das arestas 1 e custo total $k^m - 1$. Dentre as AGM's de todos os conjuntos possíveis de amostras, essa é a de custo máximo. Basta ver que, excluindo qualquer conjunto de nós da mostra, a árvore resultante tem custo menor ou igual à AGM de custo máximo, visto que a última contém a primeira.

Sendo assim, tem-se que $S_{max} = 2n+m-1+2(k^m-1)$. Logo, para qualquer conjunto de dados com custo S pela técnica, tem-se que

$$\begin{aligned} S \leq S_{max} < \overline{S} &\Leftrightarrow 2n + m - 1 + 2(k^m - 1) < m \cdot n \\ &\Leftrightarrow n(m-2) > m - 3 + 2k^m \\ &\Leftrightarrow n > \frac{m - 3 + 2k^m}{m-2} \quad . \end{aligned}$$

Note, contudo, que o minorante encontrado para n é muito conservador, visto que, normalmente, o valor k^m é extremamente alto.

4.1 Experimentos numéricos

A seguir, são descritos os experimentos numéricos de ordem sintética realizados. Seus respectivos resultados são exibidos, bem como análises são tecidas.

Sequências genéticas

Para o primeiro experimento, objetivou-se o armazenamento de sequências de bases nitrogenadas em amostras (sintéticas) de DNA. Essas sequências são compostas por quatro valores possíveis: A, G, C e T - correspondentes aos compostos químicos adenina, guanina, citosina e timina, nessa ordem. À luz da técnica de armazenamento apresentada, tal fato se traduz na quantidade de elementos presentes no conjunto universo de cada entrada das amostras, logo $k = 4$. Com iguais probabilidades, cada elemento da sequência sintética assumiu um dos valores citados. Apesar da leitura sequencial das fitas de DNA, pode-se converter o vetor original de bases em uma matriz de tamanho $m \cdot n$. No experimento, fixou-se $m = 10$, ou seja, cada vetor de amostras possui 10 entradas. Assim, gradualmente variou-se a quantidade de linhas da matriz, de modo a gerar os resultados apresentados na tabela a seguir.

Tabela 1: Dados coletados a partir do experimento com sequências genéticas.

n	Redução de armazenamento	$\hat{d}$
10	-19,0 %	5.56
100	9,3 %	4.03
1.000	34,1 %	2.80
10.000	52,7 %	1.87

Para além da quantidade de amostras n, e da redução de armazenamento resultante, também se exibe o custo médio das arestas $\hat{d}$ da AGM. Nesse experimento, vale ressaltar que o majorante para $\hat{d}$ equivalia à $\frac{1}{2}(m-2) = 4$. No primeiro registro, nota-se a superioridade do método tradicional de armazenamento - evidenciada pela "redução" negativa -, possivelmente em virtude da baixa similaridade entre as poucas 10 amostras. O valor médio do arco acima do majorante corrobora a hipótese anterior.

A partir de 10.000 amostras, destaca-se a redução de armazenamento por mais da metade. Supõe-se que a tendência de redução perpetua-se para valores de n superiores. Contudo, é justo apontar que, em razão de insuficiência computacional, não foi possível estender a escala dos experimentos sobre sequências genéticas. A saber, ultrapassa-se o limite de memória RAM disponível. No entanto, para conjuntos maiores, seria possível parcelar o armazenamento em safras. A título de exemplo, o armazenamento de 100 mil amostras poderia ser dividido em dez arquivos separados, cada um referente a subconjuntos disjuntos de 10 mil amostras. Dessa maneira, abriria-se mão de uma possível, e custosa, redução superior a 50 %, em troca de uma metodologia mais eficiente em termos de escassez computacional.

Imagens binárias

Nesse experimento, procurou-se otimizar o armazenamento de imagens binárias, isto é, imagens em que cada pixel assume apenas 2 valores: branco ou preto - 0 ou 1. No contexto de visão computacional, o conjunto de dados para treinamento e validação no aprendizado de máquinas envolve, por vezes, o processo de conversão de imagens em tons binários. Essa conversão visa ao realce de entidades mediante a separação entre objeto de interesse e fundo da imagem. Naturalmente, a redução do armazenamento de amostras que substanciam a inferência computacional possibilita a expansão dos conjuntos dessas mesmas amostras.

O experimento é munido de n imagens representadas matrizes de dimensão 28×28. Aqui, assume-se que as imagens, as quais representam uma entidade a ser discernida (letra, dígito - padrões quaisquer), possuem uma taxa de diferença máxima $p \in [\frac{1}{28}, 1]$ em cada linha de pixels - a fim de imprimir variedades sutis a figuras de uma mesma entidade. Dessa maneira, a primeira linha de cada imagem terá, no máximo, $28 \cdot p$ pixels diferentes das demais primeiras

linhas. O mesmo se aplica às linhas de pixel subsequentes. Inicialmente, gerou-se aleatoriamente a primeira imagem, em que cada entrada assumiu com igual chance o valor 0 ou 1; então a segunda imagem foi construída a partir de no máximo p alterações por linha da primeira imagem; já para a terceira, aplicaram-se no máximo p mudanças por linha sobre a segunda imagem; assim se sucedeu até a n-ésima imagem .

Sob essa perspectiva, aplica-se a técnica por AGM's, por algoritmo de Prim com fila de prioridade, a cada posição de linha das figuras, isto é, todas as i-ésimas linhas do conjunto de imagens serão salvas em uma matriz de armazenamento ótimo. Pela ótica das redes de similaridade, busca-se, para cada posição de linha i, a AGM cujos nós são as n linhas i.

Nos primeiros testes, fixou-se $p = 1/4$ e variou-se o tamanho n do conjunto, paralelamente registrando-se a redução de memória obtida. Também se computou o tempo de execução, em segundos, da aplicação da técnica. Os resultados são exibidos na Tabela 2. Para além da modificação no tamanho do conjunto de amostras, submeteu-se p à variação sob $n = 500$ fixo. Os resultados referentes a esses testes situam-se na Tabela 3.

Tabela 2: Dados coletados a partir do experimento com imagens binárias, variando-se o tamanho n do conjunto, fixada a taxa máxima de diferença entre linhas $p = 1/4$.

n	Redução de armazenamento	Tempo (s)
25	74,6 %	0,40
50	78,3 %	0,54
100	81,0 %	0,98
250	83,7 %	4,08
500	85,7 %	18,27
1.000	87,1 %	68,40

Tabela 3: Dados coletados a partir do experimento com imagens binárias, variando-se a taxa máxima de diferença entre linhas p, fixado o tamanho do conjunto $n = 500$.

p	Redução de armazenamento	Tempo (s)
1/14	95,9 %	13,10
1/8	93,3 %	14,32
1/4	85,7 %	18,27
1/2	75,3 %	19,74
3/4	68,6 %	18,53
1	64,7 %	17,48

Tal como os resultados da Tabela 2 indicam, à despeito da alta redução de ar-

mazenamento pela metodologia empregada, o elevado dispêndio computacional (tanto em termos de memória RAM, quanto em tempo de execução) inviabiliza escalar o conjunto de amostras. A título de ilustração, o MNIST *database* (*Modified National Institute of Standards and Technology database*), base de dados para reconhecimento de dígitos de 0 a 9, possui 60 mil amostras para treino, e 10 mil para teste. Sendo assim, essa quantidade excede os domínios de operação da presente implementação (serial) da técnica. Daí, supõe-se que a distribuição paralela do armazenamento de cada posição de linha possa, muito provavelmente, acelerar a compressão total. Porém, ressalta-se a alta dependência de memória RAM pela técnica - fator limitante à computação paralela. Novamente, seria viável - e aconselhado - o armazenamento das imagens em safras, a fim de particionar o tempo de execução, em detrimento da maior redução de memória necessária.

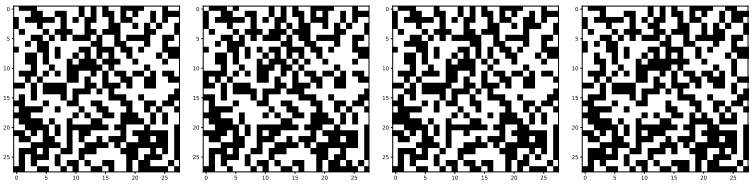

Imagem 8: Algumas imagens binárias geradas sinteticamente com taxa máxima de diferença entre linhas $p = 1/10$.

Quanto aos resultados oriundos do segundo teste, nota-se que, mesmo para o caso em que a taxa de diferenças é máxima (isto é, $p = 1$), reduz-se o armazenamento por mais da metade. Para valores de p inferiores à $1/4$, evidencia-se a integral superioridade da técnica - visto que, nesses cenários, assegura-se fortemente a hipótese inicial de baixa variabilidade entre as amostras do conjunto.

Redes Neurais Binárias

No contexto de aceleração computacional, e sobretudo de redução de armazenamento, Redes Neurais Binárias lançam mão da binarização dos pesos e ativações (+1 ou -1), reduzindo o custo das operações aritméticas, bem como limitando o espaço de cada variável a 1 bit (Courbariaux et al. (2016) [4]). Sob essa perspectiva, o armazenamento dos coeficientes finais da rede para posterior inferência prática, usufrui das propriedades necessárias para a aplicação da técnica por AGM's. No artigo, em um dos resultados de *benchmark*, utilizou-se uma Rede Binária - cuja arquitetura consistia em três camadas de 4096 unidades binárias - para a inferência de dígitos de 0 a 9, a partir do banco de dados MNIST.

Para esse terceiro experimento, procurou-se otimizar o armazenamento dos $3 \times 4096 = 12288$ coeficientes binários. Para tanto, considerou-se uma matriz de dimensões $n \times m$, tal que $n \cdot m = 12288$, cujas entradas eram -1 ou 1, com mesma probabilidade. Desse modo, variou-se ambas as dimensões a fim de analisar o perfil de redução de memória resultante. Sob a perspectiva da técnica estudada, tais variações simbolizam a variação do número de amostras n do conjunto de dados (consiste no número de nós da AGM), e o número de entradas m das amostras, mantendo fixo o valor $n \cdot m$. Complementarmente, registrou-se a razão entre o tempo médio de reconstrução das amostras do conjunto a partir da matriz de armazenamento ótimo T_R, e o tempo médio de acesso das amostras a partir da matriz original T_A.

Tabela 4: Dados coletados a partir do experimento com Redes Neurais Binárias, variando-se os valores de n e m.

n	m	Redução de armazenamento	T_R/T_A
3072	4	74,73 %	$0{,}15 \cdot 10^2$
768	16	66,06 %	$0{,}56 \cdot 10^2$
192	64	30,59 %	$1{,}16 \cdot 10^2$
48	256	12,43 %	$2{,}53 \cdot 10^2$
12	1024	4,21 %	$3{,}12 \cdot 10^2$

Com base na Tabela 4, observa-se que a otimização de armazenamento é mais frutífera mediante o acréscimo do número de amostras n, e o concomitante decréscimo do número de variáveis m para cada vetor do conjunto. Em relação aos tempos obtidos, conclui-se que T_R manteve-se na ordem de 10^{-6} s (e até mesmo 10^{-5} s para $12 \leq n \leq 48$), ao passo que T_A manteve-se constantemente na ordem de 10^{-8} s. Por tal motivo, a relação T_R/T_A alcançou ordem de 10^2 em todos os testes performados. Atesta-se que tal atraso de T_R se dá em virtude do acúmulo de passos para reconstrução das amostras, à medida que a altura da AGM aumenta. Dessa maneira, no pior dos casos, a reconstrução de um vetor é proporcional à altura da árvore.

5 Conclusão

O presente projeto reflete, não cronológica nem empiricamente, o processo de estudo e implementação da técnica de redução de armazenamento através de Árvores Geradoras Mínimas como estrutura de dados responsável por determinar caminhos de reconstrução de vetores mediante sua similaridade com o resto do conjunto de amostras. Nesse sentido, descreveu-se a sucinta base matemática da técnica; além de alguns algoritmos clássicos de busca por AGM. Quanto ao escopo computacional, foram testadas diferentes formulações de Prim e Kruskal - ambas debruçando-se sobre abordagens diretas ou rebuscadas de im-

plementação. A predileção individual levou à priorização de Prim munido de *min-heap* ao longo dos experimentos numéricos. Pertinente ao aspecto computacional do estudo, destaca-se o fato de que os programas, bem como os principais algoritmos, seguem lógicas seriais e, consequentemente, tornam-se instáveis conforme escalam-se as dimensões do problema.

No que tange à viabilidade da técnica de compressão estudada, vale apontar que os minorantes que asseguram sua superioridade apresentam duas fragilidades:

1. O primeiro minorante - relativo ao custo médio das arestas $\hat{d}$ - preconiza a construção da AGM, antes mesmo de assegurar sua viabilidade;

2. Já o segundo - referente ao número n de amostras mínimo - é intrinsecamente inflado. Basta ver que os experimentos descritos alcançam significativas reduções de armazenamento para valores de n substancialmente menores ao minorante calculado;

Esses fatores, além dos previamente citados, pesam em desfavor à técnica, pois não garantem facilmente de antemão uma real redução de armazenamento. No entanto, é justo notar a elevada redução obtida pelos experimentos no presente projeto, além daqueles realizados no artigo original. Daí, em contextos onde o poder computacional de processamento supera, significativamente, a capacidade de armazenamento, a compressão por AGM desponta como opção viável.

Implementação computacional

No link a seguir, aquele que lê o presente projeto pode acessar todos os códigos relativos à técnica de compressão, bem como aos experimentos numéricos.

`https://github.com/Beprados/Network-Flow-Project`

Referências

[1] AHUJA, Ravindra K.; MAGNANTI, Thomas L.; ORLIN, James B. Network flows: theory, algorithms, and applications. Englewood Cliffs, NJ: Prentice Hall, 1993.

[2] Kang, Lee, Chin-Liang Chang and Shi-Kuo Chang, "Storage Reduction Through Minimal Spanning Trees and Spanning Forests," in IEEE Transactions on Computers, vol. C-26, no. 5, pp. 425-434, May 1977, doi: 10.1109/TC.1977.1674859.

[3] FEOFILOFF, Paulo. Notas de aula. Disponível em <https://www.ime.usp.br/ pf/>. Acessado em 30 de maio de 2025.

[4] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1. arXiv preprint arXiv:1602.02830. https://arxiv.org/abs/1602.02830.