

UNIVERSIDADE FEDERAL FLUMINENSE

MARCIO CONTENTE ARESE

**COMPUTAÇÃO REVERSÍVEL: PRINCÍPIOS E
PROJETO DE ALGORITMOS**

NITERÓI

2026

MARCIO CONTENTE ARESE

COMPUTAÇÃO REVERSÍVEL: PRINCÍPIOS E PROJETO DE ALGORITMOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Orientador:

UÉVERTON DOS SANTOS SOUZA

NITERÓI

2026

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

A678c Arese, Marcio Contente
 COMPUTAÇÃO REVERSÍVEL : PRINCÍPIOS E PROJETO DE ALGORITMOS /
 Marcio Contente Arese. - 2026.
 125 f.

 Orientador: Uéverton dos Santos Souza.
 Dissertação (mestrado)-Universidade Federal Fluminense,
 Instituto de Computação, Niterói, 2026.

 1. Otimização combinatória (Computação). 2. Algoritmo
 computacional. 3. Programação (Computação). 4. Consumo de
 energia elétrica. 5. Produção intelectual. I. Souza,
 Uéverton dos Santos, orientador. II. Universidade Federal
 Fluminense. Instituto de Computação. III. Título.

CDD - XXX

MARCIO CONTENTE ARESE

COMPUTAÇÃO REVERSÍVEL: PRINCÍPIOS E PROJETO DE ALGORITMOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Aprovada em julho de 2026.

BANCA EXAMINADORA

Prof. UÉVERTON DOS SANTOS SOUZA – Orientador, UFF

Prof. LUIS ANTONIO BRASIL KOWADA, UFF

Prof. PEDRO HENRIQUE GONZÁLEZ SILVA, UFRJ

Niterói

2026

A Carl Sagan e a Isaac Asimov, in memoriam, formadores de meu espírito científico.

Agradecimentos

À Lilian, que sugeriu que eu cursasse o mestrado e sofreu suas consequências.

À universidade pública e, em particular, a UFF, pela possibilidade.

A Uéverton Souza, pela aposta e orientação que o tornou realidade.

A todos aqueles que pacientemente me ouviram durante a gestação do trabalho: como disse Richard Feynman, tentar explicar um assunto é a melhor forma de aprendê-lo.

Resumo

O computador digital tornou-se o dispositivo central da civilização tecnológica contemporânea, estando presente em praticamente todos os sistemas inteligentes. A lógica e a física irreversíveis que fundamentam a computação tradicional implicam dissipação de energia sempre que ocorre destruição de informação, ameaçando a continuidade do avanço da eficiência e da miniaturização. As demandas computacionais crescentes da inteligência artificial intensificam essa preocupação, traduzindo-se em custos energéticos e econômicos significativos. A computação reversível surge como alternativa potencial capaz de superar o limite inferior fundamental de dissipação associado à irreversibilidade lógica. Este trabalho apresenta as bases físicas e lógicas da irreversibilidade e da reversibilidade, revisita o desenvolvimento histórico da computação reversível e examina as condições lógicas para a irreversibilidade de programas e para a semirreversibilidade — entendida como a admissão controlada de operações irreversíveis em um arcabouço parcialmente reversível. Dá-se especial atenção à distinção entre irreversibilidade física e lógica, ao princípio de Landauer que relaciona perda de informação a custo termodinâmico e a perspectivas críticas que questionam se a dissipação é estritamente inevitável em todos os cenários práticos. Utiliza-se uma linguagem de pseudocódigo acrescida de reversibilidade para implementar algoritmos clássicos — Subsequência Comum Mais Longa (LCS), Floyd–Warshall e Radix Sort — examinando o impacto da reversibilização nas complexidades de tempo e espaço. Os resultados evidenciam tanto as restrições impostas pelas estruturas de dependência quanto a viabilidade de soluções algorítmicamente reversíveis sem custo energético assintótico adicional.

Palavras-chave: computação reversível; irreversibilidade lógica; inversão de programas; programação reversível; energia e informação.

Abstract

Digital computers have become the central device of contemporary technological civilization, embedded in virtually all intelligent systems. The irreversible logic and physics on which they are traditionally based imply energy dissipation whenever information is destroyed, threatening the continued advance of efficiency and miniaturization. The ever-growing computational demands of artificial intelligence intensify this concern, translating into substantial economic and energetic costs. Reversible computing emerges as a potential alternative capable of overcoming the fundamental lower bound on dissipation associated with logical irreversibility. This work presents the physical and logical foundations of irreversibility and reversibility, revisits the historical development of reversible computing, and examines the logical conditions for program invertibility and semireversibility—the controlled admission of irreversible operations within a partially reversible framework. Particular attention is devoted to the distinction between physical and logical irreversibility, to Landauer’s principle relating information loss to thermodynamic cost, and to critical perspectives that question whether dissipation is strictly unavoidable in all practical scenarios. A pseudocode language with added reversibility is used to implement classical algorithms — Longest Common Subsequence (LCS), Floyd–Warshall, and Radix Sort — examining the impact of reversibilization on time and space complexity. The results illustrate both the constraints imposed by dependency structures and the feasibility of constructing fully reversible algorithmic solutions without asymptotic energy cost.

Keywords: reversible computing; logical irreversibility; program inversion; reversible programming; energy and information.

Lista de Figuras

1	Local da ficha catalográfica	
2	Distribuição de microestados para cada macroestado (12 partículas em um compartimento dividido ao meio)	21
3	Interação entre os subsistemas informacional e térmico em um circuito irreversível . .	25
4	Comparação entre uma operação irreversível com descarte de informação (Landauer), uma operação reversível com cancelamento, e uma versão reversível com cópia limpa (Bennett).	28
5	Representação gráfica dos conceitos de determinismo <i>forward</i> e <i>backward</i> , assinalando os caminhos válidos e inválidos.	29
6	Portas lógicas reversíveis: CNOT (XOR/FAN-OUT), Fredkin (CSWAP), Toffoli (AND/NAND) e Toffoli generalizada.	34
7	Toffoli como porta lógica universal	35
8	Exemplo de iteração no algoritmo ingênuo: o padrão P é comparado com a janela de T que se inicia na posição 2.	36
9	Onda trapezoidal, indicando a transição suave de voltagem que permite o aproveitamento da energia.	41
10	Paradigmas utilizáveis no contexto da inversão: (a) Desfazer-Refazer-Fazer; (b) Avançar-Reverter-Confirmar; e (c) Iniciar-Reverter-Confirmar.	54
11	PISA – <i>Pendulum Instruction Set Architecture</i> [69] [2] [32].	57
12	Mecânica do salto incondicional reversível em PISA	58
13	Programa em PISA para cálculo de $N \times M$. Os registradores R1 e R2 são inicializados com os parâmetros da multiplicação. Na fase direta, R3 acumula as R2 adições de R1, onde R2 é decrementado e R4 é incrementado a cada iteração. O resultado R3 é mostrado (por duas vezes) e o passo do PC é invertido na linha 14. Na fase reversa, as operações das linhas 7 a 9, coração do <i>loop</i> , são invertidas, bem como as das linhas 1 e 2.	59

14	Sintaxe simplificada de Janus	60
15	Fluxograma dos controles de fluxo reversíveis de Janus.	60
16	Programa em Janus para cálculo de $\lfloor \sqrt{n} \rfloor$ [43].	61
17	Sintaxe de Eel nos níveis baixo, médio e alto de abstração, sendo: $x \in \text{Variáveis}$, $q \in \text{Funções}$, $c \in \text{Constantes}$	63
18	Possibilidades considerando a reversibilidade e a irreversibilidade dos diversos níveis de abstração. Os estágios em branco significam que ainda não foram implementados de forma geral, escalável e comercialmente viável.	65
19	Tradução do condicional <code>if e_1 then s_1 else s_2 fi e_2</code> para PISA.	66
20	Tradução do <code>loop from e_1 do s_1 loop s_2 until e_2</code> para PISA.	67
21	Tradução de chamada de função em PISA.	67
22	Explosão da quantidade de cálculos na solução recursiva: os recálculos de subproblemas são destacados em fundo cinza.	73
23	Matriz da programação dinâmica destacando a dependência de $LCS[i,j]$ em relação aos outros valores, que devem estar calculados previamente.	74
24	Caso básico do algoritmo de mínimo custo. p_1 é a menor distância possível entre u e k , considerando todos os outros $k - 1$ possíveis intermediários; p_2 significa o mesmo entre k e v . p é o menor caminho entre u e v , considerando todos os k possíveis vértices intermediários.	78
25	Explosão do tempo de execução pelo recálculo de subproblemas em Floyd-Warshall na abordagem <i>top-down</i> : as repetições estão destacados em fundo cinza.	80
26	Matriz da programação dinâmica destacando a dependência de $D^k[u,v]$ em relação aos outros valores, que já estão disponíveis em função do cálculo prévio de D^{k-1}	81
27	Fluxo de computações, recomputações e reversões do algoritmo reversível Floyd-Warshall com <i>checkpoints</i> para $n = 6$. Entre colchetes está a matriz original D^i que permite a computação, recomputação ($[D^i_+]$) ou reversão ($[D^i_-]$) em cada caso.	85
28	Ordenação pelo processo repetido de esvaziamento (a) e recomposição (b) do vetor usando filas (<i>queues</i> ou <i>buckets</i>), que recebem os elementos com base no valor do dígito significativo analisado na iteração.	89

29	A transformação da representação de entrada dos elementos V_{in} no vetor de ponteiros $P[p_1]$	93
30	Movimento do primeiro ponteiro para sua lista correspondente, na iteração $i_d = 1$. Os ponteiros para os elementos a serem ordenados $P[p_1]$ e as listas $P[p_2]$ podem ser visualizados, juntamente com informações auxiliares sobre contagem de elementos, pontos inserção das listas e índices de percurso das estruturas. Os destaques em cinza indicam o que está em jogo na escolha da lista de destino para um elemento e as informações que são inseridas para permitir a reversão – (a) e (b). [•] indica uma posição zerada.	96
31	Representação do <i>qubit</i> através da esfera de Bloch. Note-se que a representação do vetor em um espaço no qual os estados $ 0\rangle$ e $ 1\rangle$ são ortogonais faz com que o ângulo obtido seja a metade daquele que na esfera corresponde a θ	111

Lista de Tabelas

1	Estágios da MTR com seus produtos em cada uma das fitas. O caractere _ indica a posição da cabeça de leitura/gravação.	31
2	A tabela verdade de uma função reversível XOR/FAN-OUT embutindo: a) a função FAN-OUT; e b) a função XOR.	32
5	Quadro descritivo das possibilidades para atualização do PC em PISA, envolvendo os registradores BR e dir. A atualização do BR precede a do PC. A literatura consultada é ambígua sobre o momento da inversão de dir durante a execução da instrução RBRA. .	58
6	Resumo operacional da porta CNOT para o caso c em superposição e o alvo t não, evidenciando o emaranhamento quântico (<i>entanglement</i>).	116
7	Mapeamento de estados notáveis na porta <i>Toffoli</i> (C^2X).	118
8	Análise de Estados Notáveis na Porta <i>Fredkin</i> (CSWAP)	118

Sumário

1	Introdução	12
2	Irreversibilidade	18
2.1	Computabilidade	18
2.2	Irreversibilidade e probabilidade	20
2.3	Entropia, informação e o limite de Landauer	21
2.4	Irreversibilidade e dissipação de calor	23
2.5	Irreversibilidade e eficiência energética: futuro?	25
3	Reversibilidade	27
3.1	Histórico	27
3.2	Definição de Copenhagen	29
3.3	Máquina de Turing reversível	30
3.4	Uma visão desde os circuitos	32
3.5	Bijetivação de problemas: um exemplo	35
4	O encontro da teoria com a realidade	40
4.1	Computação energeticamente eficiente	40
4.2	Computador completamente reversível?	42
4.3	O fim da computação reversível?	43
4.4	Análise de alguns representantes	44
4.4.1	Pendulum	44
4.4.2	Máquina de Demaine et al. [12]	45

4.4.3	Ice River	46
5	Interlúdio: a inversão de programas	48
5.1	C inversível	50
5.2	Alguns paradigmas da inversão e suas aplicações	53
5.3	<i>Checkpointing</i>	55
6	Programação reversível	56
6.1	Duas estruturas: a língua de sogra e o tapete	56
6.2	Linguagens reversíveis	57
6.2.1	PISA	57
6.2.2	Janus	59
6.2.3	Eel: <i>Energy-efficient language</i>	60
6.2.4	Outras linguagens reversíveis	64
6.3	O que temos hoje	64
6.4	Análise da reversibilidade das principais estruturas de alto nível	65
6.5	Adicionando reversibilidade à linguagem de pseudocódigo	67
7	Versões reversíveis para algoritmos clássicos	70
7.1	Subsequência Comum Mais Longa (Longest Common Subsequence – LCS)	71
7.1.1	Regras de Recorrência	71
7.1.2	Solução recursiva	72
7.1.3	Programação Dinâmica	72
7.1.4	Dificuldades encontradas na reversibilização	74
7.2	Floyd-Warshall	78
7.2.1	Solução recursiva	79
7.2.2	Programação dinâmica	80

7.2.3	Dificuldades encontradas na reversibilização	82
7.2.4	Revisitando a questão do espaço	82
7.3	Radix Sort	89
7.3.1	O algoritmo clássico	89
7.3.2	Dificuldades enfrentadas na reversibilização	90
7.3.3	Sobre a bijetivação do problema e a mecânica da solução	91
7.3.4	A questão do espaço – não sem a do tempo	92
7.3.5	Pseudocódigo e análise da complexidade	95
8	Conclusão	101
	REFERÊNCIAS	104
	Apêndice A – Elementos de Computação Quântica	110
A.1	Motivação	110
A.2	O <i>qubit</i> e os estados básicos	110
A.3	Números complexos	112
A.4	Portas quânticas	113
A.4.1	Portas de <i>qubit</i> único (<i>Single-qubit gates</i>)	113
A.4.2	Portas de <i>qubit</i> múltiplo (<i>Multiple-qubit gates</i>)	115
A.5	Matrizes de permutação unitárias e produtos tensoriais	119
A.6	Programação quântica	119
A.7	Relação com a computação reversível clássica	120

1 Introdução

O computador digital ocupa a posição central no atual estágio civilizatório. Inicialmente tinha a dimensão de salas e prédios e seu uso era restrito a centros de pesquisa e de defesa; uma evolução sem precedentes, entretanto, o atingiu em velocidade, miniaturização e capacidade. Assumiu a forma de equipamento pessoal de uso geral para depois entrar em simbiose com praticamente todos os dispositivos de uso específico que se tornaram, por isso, "inteligentes". O computador está hoje na segurança dos veículos, na escolha da rota a percorrer ou de qual filme assistir, na forma como pessoas acessam outras pessoas ou as notícias do mundo – mundo que agora cabe em um bolso. Até 2030, estima-se que 65% da população global esteja conectada à internet através de *smartphones*, totalizando 5,5 bilhões de pessoas [16]. No Reino Unido, a taxa de penetração de *smart TV* alcançou quase 75% em 2023 e, na Alemanha, praticamente todas as TV vendidas são inteligentes [11].

Com a expansão massiva da computação digital, a demanda por energia elétrica cresceu e continua a crescer de forma vertiginosa. A computação consumia entre 4% e 6% da energia mundial em 2020 [55] e estima-se que tenha atingido já o consumo entre 5% e 9% [67]. O uso diário de energia do ChatGPT em 2024, por exemplo, foi estimado como equivalente ao de quase 180.000 famílias dos EUA, e uma simples interação pode envolver consumo hídrico da ordem de dezenas de centilitros [28].

A água, nesse contexto, é empregada na remoção do calor gerado pelo processamento.

Hoje, o computador é energeticamente ineficiente quando comparado aos limites físicos fundamentais, podendo ser descrito — na formulação provocativa de Frank — como uma torradeira sofisticada que realiza computação como efeito colateral [22]. O calor é o subproduto inevitável do processamento em virtude dos paradigmas lógico e físico adotados desde o início da era digital. A computação clássica se estruturou a partir de operações logicamente irreversíveis. Uma operação é dita irreversível quando

diferentes estados de entrada produzem o mesmo estado de saída, impossibilitando a reconstrução unívoca do estado anterior a partir do resultado obtido.

De Neumann e Burks [48] e, sobretudo, de Landauer [39] vem o resultado segundo o qual o apagamento lógico de informação implica em dissipação termodinâmica mínima. Landauer demonstrou que a eliminação de um bit de informação acarreta dissipação de energia da ordem de $kT \ln 2$, aproximadamente 0,018 eV ou 3×10^{-21} J por bit à temperatura ambiente. Esse valor passou a representar um limite físico fundamental associado ao descarte de informação. O mecanismo envolvido é o do apagamento e da sobrescrição de valores: ao eliminar informação antiga para substituí-la por nova, a energia associada ao estado anterior deve, no mínimo, ser dissipada.

A diminuição do tamanho dos transistores tornou-os simultaneamente mais baratos, mais rápidos e mais eficientes em termos energéticos. Tal cenário fomentou investimentos maciços na miniaturização contínua dos semicondutores. O número de dispositivos por área de chip dobrou inicialmente a cada ano e, posteriormente, a cada 18 meses, fenômeno que ficou conhecido como Lei de Moore [46]. Esse avanço sustentou ganhos de desempenho por unidade de custo que tornou tolerável, por décadas, o desperdício energético inerente ao paradigma irreversível. Contudo, a Lei de Moore não é uma lei física. A miniaturização aproxima-se de limites práticos relacionados à densidade de potência, ao aquecimento dos componentes [5] e à confiabilidade do sinal.

Esforços substanciais têm sido realizados para ampliar a eficiência energética sem abandonar a tecnologia CMOS (*Complementary Metal-Oxide-Semiconductor*) e o paradigma convencional. O custo dessas iniciativas, entretanto, é crescente. A construção de plantas de fabricação de chips de última geração atinge a ordem de dezenas de bilhões de dólares. Técnicas de microrresfriamento intrachip vêm sendo empregadas [58]. Em 2024, empresas de refrigeração líquida ampliaram drasticamente sua capacidade produtiva em resposta à expansão dos *data centers*, impulsionada pelo desenvolvimento acelerado da Inteligência Artificial (IA) [27]. Estima-se que, até 2026, a demanda energética de *data centers*, IA e criptomoedas possa duplicar em relação a 2022 [67]. Há também o aspecto ambiental: parcela significativa da energia empregada provém ainda de combustíveis fósseis, e *data centers* já respondem por fração relevante das emissões globais de gases de efeito estufa [45].

Não há consenso quanto ao momento em que os dispositivos CMOS se aproximarão efetivamente do limite de Landauer. O que se sabe é que tal limite decorre

exclusivamente do apagamento de informação, independentemente da tecnologia empregada. A reversibilidade lógica constitui, até onde se conhece, o único caminho capaz de contornar o custo termodinâmico associado à destruição de informação. Em princípio, operações lógicas implementadas de maneira reversível podem aproximar-se arbitrariamente de dissipação nula no regime adiabático ideal.

A reversibilidade lógica requer primitivas bijetoras, isto é, funções nas quais cada estado de saída corresponde a um único estado de entrada. O conjunto dessas primitivas deve preservar o mesmo poder computacional da computação tradicional.

Para que um sistema seja considerado logicamente reversível, deve-se observar o princípio da não sobrescrição e do não apagamento de valores em todos os níveis de abstração, dos circuitos à programação. Linguagens reversíveis oferecem construções específicas para inicialização, controle de fluxo e chamadas de procedimento. Programas reversíveis assumem frequentemente a forma de uma imagem especular: na primeira fase computam e acumulam informação auxiliar; após copiar o resultado desejado, realizam a descomputação, restaurando o estado inicial [6].

A manutenção de informação auxiliar implica custo de espaço adicional. No método clássico de Bennett [6], esse custo pode crescer linearmente no tempo da computação original, embora estratégias de recomputação permitam diferentes compromissos entre tempo e espaço.

Diante desse cenário, pode-se perguntar por que dedicar atenção a esse tema.

Ainda não dispomos de tecnologia plenamente reversível; contudo, mecanismos de recuperação parcial de energia vêm sendo estudados sob a denominação de computação adiabática ou energeticamente eficiente. Esses mecanismos não invalidam o princípio de Landauer, mas exploram a possibilidade de reduzir drasticamente a dissipação associada às transições lógicas [38]. Iniciativas industriais recentes, como as associadas ao protótipo Ice River desenvolvido pela *startup* Vaire Computing (vaire.co) em colaboração com o acadêmico Michael P. Frank, indicam interesse concreto na exploração desses princípios. O nível de recuperação de energia anunciado já alcança 50% [25].

Mesmo que atualmente haja obstáculos na construção de um computador perfeitamente reversível devido às interações inevitáveis com um ambiente termodinamicamente irreversível [42], a disciplina permanece fundamental. Ela estabelece limites inferiores para a dissipação associada à perda de informação e fornece ferramentas

algorítmicas para analisar o custo estrutural da reversão de computações.

Isto posto, embora este trabalho aponte algumas questões na implementação física do paradigma reversível, partimos dele. O foco principal do trabalho é o de dar um passo significativo no campo da análise e do projeto de algoritmos quanto à reversibilidade. Dito de outra forma, nosso interesse é, assumido o paradigma, apresentar seus princípios e discutir como implementar estratégias clássicas e fundamentais respeitando as regras da reversibilidade. Mais precisamente, iremos discutir a reversibilidade de algoritmos clássicos para três problemas fundamentais: o de achar a subsequência mais longa comum a duas sequências ou *strings*, o de computar o caminho mínimo entre todos os pares de vértices de um grafo direcionado ponderado, e o da ordenação.

O segundo capítulo deste trabalho aborda as bases físicas da computação reversível, a origem da irreversibilidade no mundo macroscópico, o princípio de Landauer e os impasses enfrentados pelo paradigma irreversível para continuar avançando em eficiência energética.

O terceiro capítulo visita o histórico lógico da reversibilidade computacional, reversibilidade como propriedade da matéria em nível microscópico, as contribuições inaugurais de Landauer e Bennett, a prova do poder de computação da Máquina de Turing reversível, e um algoritmo de transformação (encapsulamento) de função irreversível em reversível. Ainda, são apresentadas as portas lógicas reversíveis mais importantes.

O quarto capítulo confronta a teoria com a realidade, mostrando o que hoje se considera possível no âmbito da computação reversível. São apresentadas as principais características de algumas arquiteturas em relação à reversibilidade: as máquinas teóricas Pendulum [69], de Demaine et al. [12] e do protótipo Ice River.

O quinto capítulo investiga a inversão de programas como uma versão mais frouxa da reversibilidade, sem o objetivo de economia de energia, e que se faz presente em algumas aplicações, como a facilidade *undo*.

O sexto capítulo trata da programação reversível. Olhamos de perto algumas linguagens reversíveis, a partir das quais analisamos a reversibilidade das estruturas de controle de fluxo. Tais análises servem de referência para a inclusão de recursos de reversibilidade na linguagem de pseudocódigo a ser utilizada no capítulo seguinte.

O sétimo capítulo apresenta nossas versões reversíveis comentadas para os algoritmos clássicos da Sequência Comum Mais Longa (*Longest Common Subsequence*),

Floyd-Warshall e Radix Sort.

O algoritmo de Floyd-Warshall calcula os caminhos mínimos entre todos os pares de nós em tempo $\Theta(n^3)$ e espaço $\Theta(n^2)$. Frank e Knight Jr. [23] argumentam que o algoritmo de Floyd-Warshall pode ser adaptado para execução reversível com espaço $\Theta(n^3)$. Demaine *et al.* [12] observaram que esse é um aumento substancial de espaço para tornar o programa reversível e, assim, economizar energia. Esses autores apresentaram uma demonstração da afirmação de Frank e Knight Jr. [23] de que solucionar o problema de CAMINHO MÍNIMO ENTRE TODOS OS PARES por meio da abordagem de multiplicação de matrizes ($\min, +$) é possível através de um algoritmo reversível que é executado em tempo $\Theta(n^3 \log n)$ e espaço $\Theta(n^2 \log n)$.

Demaine *et al.* [12] levantaram a seguinte questão:

Existe um algoritmo reversível para o problema de caminho mínimo entre todos os pares com sobrecargas de tempo e espaço apenas por um fator constante?

Para lidar com isso, eles apresentaram um algoritmo semirreversível de energia reduzida para o problema. Segundo Demaine *et al.* [12], um algoritmo totalmente reversível ainda parece difícil de alcançar; de modo geral, os algoritmos de caminho mínimo parecem constituir uma categoria difícil de tornar reversível, pois utilizam muito pouco espaço e fazem uso frequente da sobrescrita de valores antigos. Até onde sabemos, existem dois algoritmos reversíveis para o problema de caminhos mínimos entre todos os pares na literatura:

- o algoritmo de Floyd-Warshall reversível proposto em [23], com complexidade de tempo $\Theta(n^3)$ e complexidade de espaço $\Theta(n^3)$;
- a abordagem baseada em multiplicação de matrizes ($\min, +$) analisada em [12], com complexidade de tempo $\Theta(n^3 \log n)$ e complexidade de espaço $\Theta(n^2 \log n)$.

Neste trabalho, e com relação à questão suscitada em [12], contribuímos com a apresentação de uma versão reversível para o algoritmo de Floyd-Warshall com complexidade de tempo $\Theta(n^3)$ e complexidade de espaço $\Theta(n^{2.5})$. Nosso método é o primeiro algoritmo reversível proposto na literatura para calcular os custos de caminhos mínimos entre todos os pares em um grafo ponderado sem ciclos negativos, executando em

tempo $O(n^3)$ e consumindo espaço $o(n^3)$. Assim, demonstramos que é possível preservar a complexidade de tempo do algoritmo clássico não reversível ao custo de apenas um aumento sublinear na complexidade de espaço (por um fator de, no máximo, $\sqrt{n}$).

2 Irreversibilidade

What's done, is done.

– *Macbeth*, ato 3, cena 4, Shakespeare

2.1 Computabilidade

O que exatamente significa computar? A resposta conduz à noção de algoritmo: uma sequência finita e bem definida de operações elementares destinadas à obtenção de um resultado [1]. Em 1937, Turing [62] apresentou um modelo abstrato que formaliza esse conceito independentemente de qualquer implementação física específica. Aquilo que é computável é aquilo que pode ser produzido por um procedimento finito executado por uma máquina abstrata.

Essa máquina, posteriormente denominada Máquina de Turing (MT), consiste em uma fita potencialmente infinita dividida em células, um cabeçote de leitura/escrita e uma unidade de controle. O alfabeto utilizado exclui o símbolo branco, que representa ausência de dado. A máquina opera localmente: em cada passo lê o símbolo sob o cabeçote, pode sobrescrevê-lo, move-se uma posição à esquerda, à direita ou permanece estacionária, e alcança um novo estado.

A dinâmica da máquina é descrita por quintuplas da forma:

[estado atual, lê, escreve, move, próximo estado]

- *estado atual*: é o estado em que se encontra a máquina; pode ser o estado inicial ou um dos estados intermediários. Como a máquina é determinística, pode haver mais de uma tupla com o mesmo *estado atual*, desde que não tratem do mesmo caracter lido (não pode haver ambiguidade em relação ao próximo estado);
- *lê*: é o conteúdo da posição da fita que a máquina está considerando no instante, que pode ser um dos caracteres do alfabeto, uma célula vazia (p.ex., β – de *branco*), ou um símbolo significando qualquer caracter do alfabeto ($\forall$);

- *escreve*: é o caracter do alfabeto que a máquina deve escrever na posição considerada ou símbolo para não escrita (p.ex., /);
- *move*: informa se a máquina deve passar a considerar a posição da direita (+), da esquerda (−), ou se deve continuar considerando a mesma posição da fita (0); e
- *próximo estado*: é o estado alcançado pela máquina após a execução dos comandos de leitura, escrita e movimentação contidos no estado atual. Pode ser qualquer estado, inclusive o estado final – quando a máquina, então, para.

Como exemplo, as tuplas necessárias para que uma MT inverta os dígitos de um número binário. As duas primeiras tuplas posicionam a máquina para considerar o início do número (da esquerda para a direita). As duas seguintes realizam a inversão e a última detecta o final do número, posiciona a máquina sobre o último dígito e termina a execução.

$$(E_1, \forall, /, -, E_1)$$

$$(E_1, \beta, /, +, E_2)$$

$$(E_2, 0, 1, +, E_2)$$

$$(E_2, 1, 0, +, E_2)$$

$$(E_2, \beta, /, -, E_F)$$

O determinismo exige que não haja ambiguidade: para cada par (*estado atual*, *lê*), no máximo uma transição é permitida.

A MT universal, concebida pelo próprio Turing, elevou o modelo a outro patamar: passou a ser possível codificar sobre a fita tanto os dados quanto a descrição de outras máquinas, permitindo a execução de programas arbitrários [1]. Essa ideia fundamenta a noção moderna de máquina de propósito geral.

A tese de Church–Turing estabeleceu a equivalência entre computabilidade efetiva e computabilidade por MT [63]. Assim, a MT tornou-se o modelo formal que inspira a arquitetura dos computadores clássicos.

Contudo, o modelo padrão de MT permite sobrescrição de células sem qualquer mecanismo de preservação do estado anterior. Essa característica introduz uma assimetria temporal: a operação é funcionalmente determinística no sentido direto, mas não reversível no sentido lógico.

2.2 Irreversibilidade e probabilidade

Muitos fenômenos do nosso dia a dia nos permitem reconhecer facilmente o sentido da passagem do tempo. Esses fenômenos são chamados de irreversíveis. Através deles somos capazes de perceber que do presente seguiremos para o futuro e não para o passado. Se não fosse assim, pipoca viraria milho e galinhas virariam pintinhos. O tempo segue um sentido único, do passado para o futuro. Sendo a irreversibilidade um aspecto tão geral do nosso dia a dia, seria de esperar que existisse uma lei fundamental da física, válida para qualquer tipo de sistema, que explicasse isso. Entretanto, as leis físicas mais fundamentais, aquelas que descrevem as interações entre as partículas e campos que conhecemos, não distinguem passado e futuro. [47].

A reversibilidade é uma das características mais fundamentais e universais da física moderna, incluídos aí os eventos quânticos [20]. Dado um estado presente é teoricamente possível reconstruir o passado porque microscopicamente os estados evoluem de forma determinística e sem perda de informação. Sendo assim, por que nos deparamos então com fenômenos que se apresentam incontornavelmente como irreversíveis?

Consideremos as distribuições possíveis de 12 partículas individualmente identificáveis ($p_1 \dots p_{12}$) em um compartimento retangular dividido imaginariamente ao meio. Definimos macroestado como o número de partículas no lado direito, e microestado como uma configuração específica das partículas. Por fim, definimos multiplicidade (Ω) como o número de microestados que satisfazem às condições de um macroestado. No exemplo, temos 13 macroestados, M_0 a M_{12} , e o macroestado M_1 tem multiplicidade de 12 microestados, uma vez que cada ocorrência corresponde a p_i estando do lado direito e o restante das partículas do lado esquerdo, i variando de 1 a 12. A Figura 2 apresenta a multiplicidade de cada macroestado do exemplo dado.

A fórmula que fornece a multiplicidade é a da combinação do número total n de partículas tomados em grupos formados por d partículas no lado direito:

$$\Omega(n,d) = \frac{n!}{d!(n-d)!}$$

Mesmo com apenas 12 partículas, a multiplicidade cresce rapidamente até atingir máximo em $d = n/2$. Para sistemas macroscópicos ($n \approx 10^{23}$), a concentração de probabilidade em torno do equilíbrio é esmagadora.

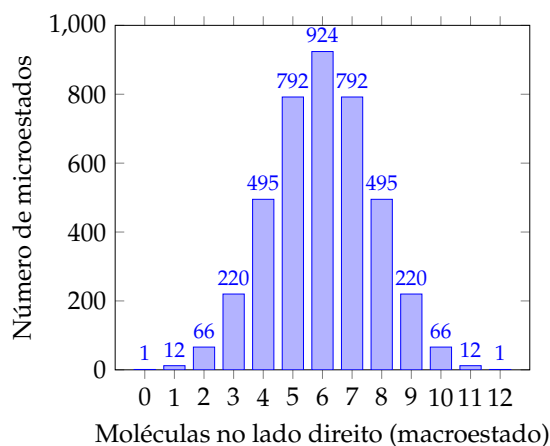


Figura 2: Distribuição de microestados para cada macroestado (12 partículas em um compartimento dividido ao meio)

Todos os microestados são igualmente prováveis, mas os macroestados não são: macroestados de alta multiplicidade são muito mais prováveis. A irreversibilidade macroscópica não decorre de impossibilidade fundamental de reversão, mas da extrema improbabilidade prática de retorno espontâneo a um macroestado de baixa multiplicidade. A informação microscópica permanece, mas torna-se inacessível.

A seta do tempo é então um efeito estatístico, uma vez que o estado de equilíbrio final alcançado pelos fenômenos macroscópicos corresponde à evolução natural do sistema, que parte de estados menos prováveis em direção àqueles mais prováveis – esta sendo uma das formas de enunciação da segunda lei da termodinâmica. Macroestados de baixa multiplicidade são chamados de menor entropia e apresentam maior organização, e vice-versa.

2.3 Entropia, informação e o limite de Landauer

A irreversibilidade computacional não é uma necessidade lógica intrínseca à computação. Trata-se, historicamente, de uma escolha arquitetural: descartar informação simplifica o controle e o gerenciamento de estados. O esquecimento daquilo que não será mais necessário torna o avançar mais leve e fácil.

As portas lógicas clássicas ilustram essa assimetria. NOT é reversível. AND, OR e NAND não são. No caso de NAND, múltiplos pares de entrada conduzem ao mesmo resultado 1. A função não é bijetora.

x	y	$\overline{x \cdot y}$
0	0	1
0	1	1
1	0	1
1	1	0

Contudo, o ponto central não é a não-bijetividade em si, mas o apagamento físico associado à implementação¹.

Landauer demonstrou que a operação lógica de apagamento — isto é, a redução do espaço de estados possíveis de um sistema físico — implica dissipação mínima de energia:

$$\Delta E \geq kT \ln 2$$

onde k é a constante de Boltzmann ($\approx 1,38 \times 10^{23}$ J/K).

Quando afirmamos que o apagamento corresponde à redução do espaço de estados possíveis, estamos dizendo que múltiplas configurações fisicamente distinguíveis passam a ser mapeadas para uma única configuração lógica. Antes da operação, o sistema poderia estar em qualquer um dentre vários estados compatíveis; após ela, essa distinção é perdida. Há, portanto, uma contração do conjunto de estados fisicamente acessíveis ao subsistema informacional.

Sob uma descrição estatística, essa contração pode ser associada à redução da incerteza quanto ao estado do sistema. Na teoria da informação, tal incerteza é medida pela entropia de Shannon [57], definida por:

$$H(x) = -\sum_i p_i \log_2 p_i$$

onde p_i representa a probabilidade atribuída a cada estado possível. Se a distribuição considerada corresponde a estados físicos reais do sistema, a diminuição dessa entropia informacional coincide formalmente com uma redução da entropia do subsistema que codifica a informação.

Poder-se-ia, então, tentar estimar a dissipação energética de uma porta lógica por meio da variação da entropia de Shannon entre suas entradas e saídas. Contudo, tal

¹Veremos no próximo capítulo que uma função não bijetiva pode ser incorporada a uma transformação ampliada e bijetiva, preservando a informação.

procedimento conduz a valores que dependem da distribuição estatística assumida para as entradas. No caso de uma porta NAND, por exemplo, o cálculo baseado em entradas equiprováveis levaria a uma variação diferente de um bit, o que poderia sugerir uma dissipação distinta daquela usualmente associada ao apagamento de um único bit.

- na entrada, $p_0 = p_1 = \frac{1}{2}$, então $H_e = -2 \cdot (\frac{1}{2} \log_2 \frac{1}{2}) = 2$ bits
- na saída, $p_0 = \frac{1}{4}; p_1 = \frac{3}{4}$, então $H_s = -(\frac{1}{4} \log_2 \frac{1}{4} + \frac{3}{4} \log_2 \frac{3}{4}) = 0,811$ bits
- $\Delta H = H_e - H_s = \boxed{1,189 \text{ bits}}$

O princípio de Landauer, entretanto, não se fundamenta na estatística de uso lógico da operação, mas na contração física efetiva do espaço de estados do sistema. Ele afirma que toda operação que reduz o número de estados físicos possíveis — isto é, que realiza apagamento lógico — deve implicar dissipação mínima de energia de pelo menos $kT \ln 2$ por bit apagado, independentemente da frequência com que determinados padrões de entrada ocorram. A coincidência formal com a entropia de Shannon surge apenas quando esta descreve probabilidades físicas reais; não se trata de uma derivação direta da teoria da informação.

Assim, a dissipação mínima não é determinada pela entropia informacional calculada a partir de uma distribuição arbitrária, mas pelo número de graus de liberdade físicos efetivamente comprimidos pela implementação da operação lógica. É essa compressão estrutural — e não a estatística simbólica — que fundamenta o limite termodinâmico da computação irreversível²

Uma vez que a implementação física é soberana para decidir o cálculo da energia dissipada, devem ser considerados também, para maior precisão, possíveis descartes nos estados internos das operações lógicas que representam decisões tomadas no nível da microarquitetura dos circuitos.

2.4 Irreversibilidade e dissipação de calor

Neste ponto, nos cabe questionar por que e como acontece a dissipação de calor na destruição de informação. Sobre isso, Padavic-Callaghan [50] diz:

²O cálculo é $\Delta H = kT \log(S_{in}/S_{out}) = kT(\log S_{in} - \log S_{out})$, onde S_{in} é a quantidade de estados possíveis na entrada etc.

Para ser eficaz, apagar informações deve ser um processo irreversível – só faz sentido se você não puder restaurar os bits apagados – mas, estritamente falando, as leis da termodinâmica proíbem isso. Elas estipulam que a informação não pode desaparecer, deve ir para algum lugar.

No entanto, se a reversibilidade é uma característica fundamental da natureza, informação não é destruída no sentido físico fundamental; ela é dispersa nos graus de liberdade microscópicos do ambiente. Isso corrobora o que dissemos acima sobre a evolução de um sistema. No caso dos circuitos, os estados que codificam os bits 0 e 1 são macroestados de baixa entropia, organizados, cuja degradação não ocorre em direção ao equilíbrio uma vez que há um investimento contínuo do sistema para os manter. Quando a informação do bit é apagada ou sobrescrita o que acontece é que esse investimento é retirado; o sistema é forçado a um único macroestado lógico, independentemente da configuração anterior. Padavic-Callaghan continua:

Então quando um bit é apagado, embora o computador se certifique de que a informação não possa ser praticamente recuperada, é impossível destruí-la completamente. Em vez disso, a informação se deteriora em aleatoriedade, ou em ruído térmico, que aumenta a entropia do computador e o faz esquentar.

Para se livrar de um bit de informação os computadores devem fazer com que a matéria que representa internamente esse bit deixe de se apresentar na forma dos estados lógicos válidos da máquina. Quando isso ocorre, dizemos que a informação foi expulsa do subsistema informacional ou mecânico e passou a habitar o subsistema termodinâmico ou térmico. Mas a abertura dessa janela entre os dois mundos tem suas consequências. Como exposto por Fredkin e Toffoli [24], a janela é uma via de mão-dupla: no sentido do subsistema térmico flui a energia livre da informação degradada na forma de dissipação de calor; mas, na via inversa, o sistema mecânico sofre os efeitos do ruído térmico, ou seja, as partículas energizadas e em movimento aleatório do estado desorganizado funcionam como um "barulho" que é mais alto quanto maior sua energia, de modo a causar erros de leitura dos estados válidos do subsistema informacional. Para manter a confiabilidade da máquina, a energia do sinal deve ser muito maior do que a energia do ruído térmico ($E_{sig} \gg kT$), e esforços devem ser feitos para expulsar as altas temperaturas das cercanias (ver Figura 3).

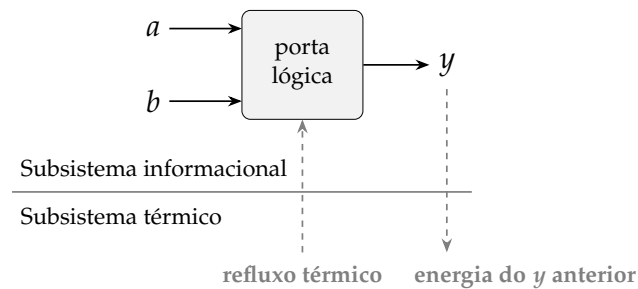


Figura 3: Interação entre os subsistemas informacional e térmico em um circuito irreversível

2.5 Irreversibilidade e eficiência energética: futuro?

A evolução da tecnologia dos semicondutores permitiu que, durante décadas, a redução do tamanho dos transistores fosse acompanhada de seu barateamento, aumento de velocidade e melhora na eficiência energética, isto é, diminuição da energia necessária para executar uma mesma quantidade de instruções de máquina. Contudo, essa trajetória passou a encontrar limites físicos e arquiteturais. A eficiência atualmente melhora mais por engenharia de sistema — paralelismo, múltiplos núcleos, especialização de *hardware* — do que por mudanças nos princípios fundamentais da computação.

A miniaturização trouxe desafios importantes. O efeito de canal curto reduz a capacidade de controle eletrostático do transistor, permitindo correntes de fuga mesmo quando o dispositivo deveria estar desligado. Além disso, a redução da espessura das barreiras físicas intensifica o tunelamento quântico, dificultando a manutenção estável dos estados lógicos. A diminuição da tensão de alimentação, utilizada para conter dissipação dinâmica, aproxima os níveis que definem 0 e 1, aumentando a suscetibilidade a ruído térmico e variabilidade.

Como consequência, tornou-se inviável continuar aumentando indefinidamente a densidade de operações por área de chip, sobretudo devido ao calor gerado pela dissipação. A integridade dos estados lógicos passa a depender da manutenção de uma razão suficientemente alta entre energia do sinal e energia térmica ambiente.

Em princípio, poder-se-ia buscar reduzir a energia do sinal. Contudo, quanto menor essa energia, maior a probabilidade de erro induzido por flutuações térmicas. Embora o limite termodinâmico fundamental associado ao apagamento lógico seja da ordem de $kT \ln 2$, a confiabilidade prática exige energias de sinal várias ordens de grandeza acima desse valor. Assim, mesmo que a engenharia consiga reduzir o número de portadores de carga ou otimizar materiais, a lógica irreversível encontra um platô de eficiência: a dissipação associada ao descarte de informação não pode ser eliminada,

apenas administrada.

Nesse contexto, a perspectiva defendida por Frank [20, 21] permanece atual: ganhos indefinidos de eficiência somente seriam possíveis mediante a adoção de lógica reversível combinada a implementações físicas capazes de recuperar, ao menos em grande parte, a energia que hoje é dissipada como calor. Em princípio teórico, essa recuperação poderia aproximar-se do total.

3 Reversibilidade

"Now, here, you see, it takes all the running you can do, to keep in the same place."

— Lewis Carroll, *Through the Looking-Glass* (1871)

3.1 Histórico

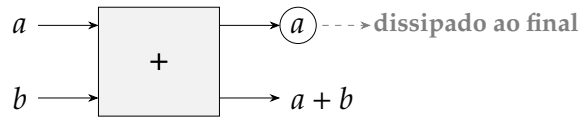
A descoberta de Landauer de que há um limite físico para a computação convencional foi a motivação original para o desenvolvimento da computação reversível [26]. Se a irreversibilidade lógica tem como consequência inexorável a irreversibilidade física, se o descarte de um bit de informação dá origem à dissipação de uma certa quantidade mínima de energia (da ordem de 10^{-21} J à temperatura ambiente), essa dissipação poderia ser evitada se de alguma forma a computação mantivesse um histórico das informações em vez de descartá-las. Landauer argumentou, contudo, que simplesmente acumular histórico representaria apenas o adiamento do descarte final inevitável, pois a memória teria de ser eventualmente reinicializada. Bennett, porém, percebeu que o histórico poderia ser reciclado pela aplicação das funções inversas daquelas primeiramente executadas¹. Esse movimento de computar e descomputar, que a princípio mostrou sua inutilidade – por somente promover o retorno ao estado inicial de uma execução com saldo zero –, foi aperfeiçoado pelo próprio Bennett com a inclusão de uma operação de cópia do resultado computado, posicionada imediatamente antes de ser iniciada a reversão ou reciclagem. O paradigma ficou conhecido como *compute-copy-uncompute* [6].

A Figura 4 mostra o percurso das ideias que acabamos de descrever, tomando como exemplo a transformação reversível $(a, b) \mapsto (a, a + b)$, que preserva a e atualiza o segundo registrador com a soma. A operação admite reversão pela aplicação da transformação inversa $(a, a + b) \mapsto (a, b)$. Antes da reversão, uma operação de soma de

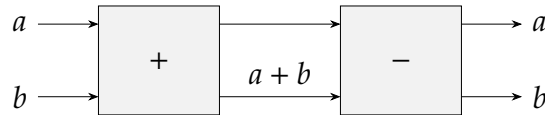
¹Tal procedimento havia sido apresentado já por Lecerf [40], trabalho que Bennett desconhecia.

$a + b$ com zero é utilizada para copiar o resultado $a + b$ ².

a) Landauer



b) 1º Bennett: reversibilidade inútil



c) 2º Bennett: computa-copia-reverte

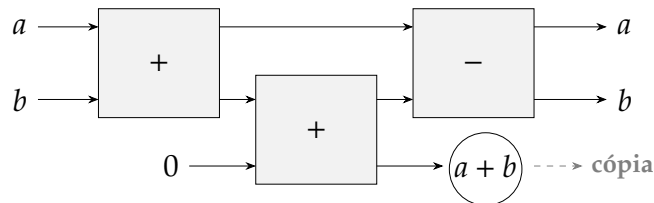


Figura 4: Comparação entre uma operação irreversível com descarte de informação (Landauer), uma operação reversível com cancelamento, e uma versão reversível com cópia limpa (Bennett).

Ao argumento da não dissipação soma-se outro, defendido por autores como Frank e Knight Jr [23] e Fredkin e Toffoli [24], o de que o modelo abstrato da computação deve refletir os princípios físicos fundamentais, incluída aí a reversibilidade das leis dinâmicas microscópicas. Como reforço dessa posição, Frank [21] afirma:

Todas as nossas leis mais fundamentais da física dinâmica microscópica são reversíveis, o que significa que, se você tivesse conhecimento completo do estado de qualquer sistema fechado em algum momento, e dos valores de todas as constantes físicas relevantes, você sempre poderia, conceitualmente, aplicar as leis da física ao contrário e determinar o estado passado do sistema em qualquer momento anterior com exatidão. (Isso é verdade até mesmo na mecânica quântica, se você soubesse o estado quântico exato do sistema.) Como consequência, é impossível ter uma situação em que dois estados detalhados possíveis diferentes em algum momento anterior pudessem ambos evoluir para se tornarem exatamente o mesmo estado detalhado um do outro em algum momento posterior, pois isso significaria que o estado anterior não poderia ser determinado exclusivamente a partir do posterior. Em outras palavras, no chão da física, a informação não pode ser destruída.

²Caso o sistema venha a reinicializar memória ou liberar registradores ao ambiente, tal descarte implicará dissipação. A interação entre um computador reversível e o ambiente é tratada na seção 4.2.

3.2 Definição de Copenhagen

Uma função é dita reversível se é ao mesmo tempo determinística para frente e para trás no tempo (*forward and backward deterministic*). Dada uma função $f : X \rightarrow Y$, f é determinística se $\forall x_i \in X, \exists! y \in Y \mid f(x_i) = y$; é determinística para trás, ou injetiva, se $\forall x_i, x_j \in X, x_i \neq x_j \Rightarrow f(x_i) \neq f(x_j)$. (ver Figura 5).

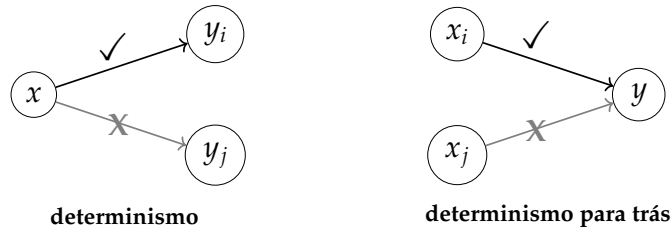


Figura 5: Representação gráfica dos conceitos de determinismo *forward* e *backward*, assinalando os caminhos válidos e inválidos.

Como toda função injetiva estabelece uma bijeção se restringirmos o contradomínio à sua própria imagem ($f(X)$), a definição de Copenhagen é matematicamente equivalente a uma bijeção sobre o espaço de estados alcançáveis [22]. Como condição para a reversibilidade utilizaremos essa definição de bijetividade.

Como circuitos e programas são composições de funções, a interpretação conhecida como de Copenhagen para a reversibilidade sustenta que se trata de um fenômeno local. Assim, uma composição é reversível se cada uma das funções que a compõem satisfizer simultaneamente o determinismo *forward* e *backward* e em qualquer granularidade analisada [34].

Se $p : W \rightarrow Z$ resulta da composição $p = g \circ f \circ e$, com $e : W \rightarrow X$, $f : X \rightarrow Y$ e $g : Y \rightarrow Z$, então a existência da inversa $p^{-1} : Z \rightarrow W$ exige que existam as inversas e^{-1} , f^{-1} e g^{-1} , sendo³ $p^{-1} = e^{-1} \circ f^{-1} \circ g^{-1}$.

Decorre da definição que a reversão de composições é a composição de reversões. Se a reversibilidade local não for observada é possível que se obtenha uma inversão semântica de p , mas sem a garantia da reversibilidade que almejamos.

³Recorde que $(f \circ g)(x) = f(g(x))$.

3.3 Máquina de Turing reversível

Bennett [6] mostrou que qualquer MT determinística pode ser simulada por uma MT reversível (MTR) com a utilização de 3 fitas. A ideia central é a de preservar informação suficiente sobre o percurso da computação para que cada passo possa ser invertido. Para isso, são necessárias algumas transformações sobre as tuplas:

1. Devem ser adicionadas quintuplas $(A_1, \beta, \beta, +, A_2)$ e $(A_{f-1}, \beta, \beta, 0, A_f)$, para marcar, respectivamente, o início e o fim da execução. Os estados A_1 e A_f não aparecem em outras quintuplas;
2. As quintuplas devem ser alteradas para considerar 3 fitas, de modo que (E_i, l, e, m, E_j) se tornará $(E_i, [l_1 \dots l_3], [e_1 \dots e_3], [m_1 \dots m_3], E_j)$, onde os componentes adicionais permitirão registrar o histórico da computação;
3. Como a ordem usual da MT é ler, escrever e depois mover, a reversão exige a reorganização dessa dinâmica. Na fase direta, a fita 2 deve se movimentar antes de ler e escrever para permitir o armazenamento do histórico. Cada quintupla $(E_i, [l_1 \dots l_3], [e_1 \dots e_3], [m_1 \dots m_3], E_j)$ é transformada, assim, em duas quádruplas:

$$\begin{aligned} &(E_i, [l_1 \dots l_3], [em_1 \dots em_3], E'_k) \\ &(E'_k, [l'_1 \dots l'_3], [em'_1 \dots em'_3], E_j) \end{aligned}$$

onde em_i representa ou um símbolo escrito ou um comando de movimentação, conforme o caso. Decorre que, se l_i é símbolo do alfabeto, então l'_i não o será, e vice-versa. O estado intermediário E'_k é de ocorrência única e garante que a execução sequencial das duas quádruplas reproduza o efeito da quintupla original na fase direta.

Na prática, para cada quintupla da MT original são geradas duas quádruplas da forma:

$$\begin{aligned} &(E_i, [l, /, \beta], [e, +, \beta], E'_k) \\ &(E'_k, [/ , \beta, /], [m, k, 0], E_j) \end{aligned}$$

A 2ª fita passa a registrar o estado k , funcionando como histórico suficiente para possibilitar a reversão;

4. Devem ser gerados os pares de quádruplas correspondentes à fase de reversão, obtidos pela inversão sistemática das transições da fase direta: troca-se estado inicial por final, símbolo lido por escrito e movimento por seu inverso:

$$(R_i, [/k,/], [m^{-1}, \beta, 0], R'_k)$$

$$(R'_k, [e,/ \beta], [l, - \beta], R_j)$$

5. Além dessas quádruplas, Bennett introduz um conjunto adicional de transições para copiar o resultado computado para a 3ª fita antes de iniciar a reversão, implementando o paradigma *compute-copy-uncompute*.

A Tabela 1 apresenta os estágios da MTR e o conteúdo das 3 fitas após cada fase. O símbolo de sublinhado indica a posição da cabeça de leitura/escrita. O determinismo para frente é herdado da MT original; o determinismo para trás decorre do fato de que a informação sobre o percurso computacional é preservada na fita de histórico, tornando a evolução global bijetiva.

Estágio	1ª fita	2ª fita	3ª fita
	<u>_</u> ENTRADA	_	_
Computa	<u>_</u> SAÍDA	<u>_</u> HISTÓRICO	_
Copia	<u>_</u> SAÍDA	HISTÓRICO <u>_</u>	<u>_</u> SAÍDA
Reverte	<u>_</u> ENTRADA	_	<u>_</u> SAÍDA

Tabela 1: Estágios da MTR com seus produtos em cada uma das fitas. O caractere _ indica a posição da cabeça de leitura/gravação.

Com recursos ilimitados, uma MTR pode computar qualquer função bijetora computável. Embora esse conjunto seja um subconjunto próprio das funções computáveis, não há limitação prática de expressividade: qualquer função computável f pode ser embutida em uma transformação bijetora da forma $x \mapsto (x, f(x))$, permitindo sua simulação reversível com aumento desprezível no tempo e aumento significativo no espaço [26].

Bennett [7] apresentou posteriormente um método para redução do espaço utilizado por computações reversíveis, baseado no chamado jogo do seixo (*pebble game*). A ideia é particionar a computação em segmentos e reverter e recomputar seletivamente históricos intermediários, em vez de acumulá-los até o final da fase direta. Com isso, a complexidade de espaço passa a depender do consumo de cada segmento multiplicado pelo número máximo de segmentos simultaneamente representados no histórico. Por seu turno, as recomputações podem afetar a complexidade de tempo.

3.4 Uma visão desde os circuitos

Toffoli [61] formulou um modelo computacional baseado na combinação de primitivas lógicas reversíveis que executam funções lógicas finitas, formando um circuito finito e sem ciclos. Concluiu que essa rede combinacional será reversível se for construída com primitivas reversíveis.

As primitivas reversíveis que propôs foram derivadas da composição das tabelas verdade de funções irreversíveis. Melhor explicado, a tabela verdade de uma função reversível pode encapsular uma ou mais funções irreversíveis.

Por exemplo, o FAN-OUT, o mapeamento de um mesmo estado lógico para mais de um nó lógico distinto, produzindo múltiplas instâncias dependentes do mesmo valor, não é aceito como bloco da reversibilidade pois que a função que executa não é bijetiva⁴. Da mesma forma, a função *ou-exclusivo* (XOR) não é reversível, pois colapsa quatro estados de entrada em apenas dois estados de saída.

Entretanto, essas duas funções podem ser obtidas a partir de uma tabela verdade única, conforme a Tabela 2:

XOR/FAN-OUT				a) FAN-OUT				b) XOR			
x_1	x_2	y_1	y_2	c	x	y_1	y_2	x_1	x_2	y	g
0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	0	1	1	1
1	0	1	0	1	0	1	0	1	0	1	0
1	1	0	1	1	1	0	1	1	1	0	1

Tabela 2: A tabela verdade de uma função reversível XOR/FAN-OUT embutindo: a) a função FAN-OUT; e b) a função XOR.

A obtenção da função FAN-OUT reversível necessita da adição um bit auxiliar c na entrada, chamado de *fonte* (*source*), e que este assuma o valor zero. Já o XOR reversível necessita da adição de um bit auxiliar g na saída, conhecido como *resíduo* (*sink*), que traduz o não descarte da variável de entrada x_2 . Completando a terminologia de Toffoli, os outros bits de entrada constituem os *argumentos* e os de saída, *resultados*. O *resíduo* é chamado de *lixo* (*garbage*) quando não é utilizado como entrada de uma outra operação.

Sendo $B = \{0,1\}$, sobre uma função finita $\phi : B^m \rightarrow B^n$, com $m > n$, que se quer reversível, é possível operar três funções. Utilizaremos como exemplo $\phi = \text{AND}$:

- primeiramente, aplica-se uma função codificadora $\mu : B^m \rightarrow B^r \times B^m$ que adiciona

⁴O domínio e o contradomínio possuem cardinalidade diferentes (2 vs. 4).

x_1	x_2	$x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

$r \leq n$ fontes inicializadas em zero à tabela verdade de ϕ : $\mu(x) = (0^r, x)$. Temos, para $r = 1$:

c	x_1	x_2	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1

- a segunda, $f : B^r \times B^m \rightarrow B^n \times B^{r+m-n}$, adiciona a bijetividade pela introdução de quantos resíduos forem suficientes para igualar a quantidade de entradas e saídas. Mas como dispor os valores? Se chamamos de X o lado esquerdo da tabela verdade de ϕ , e de Y o lado direito, temos a seguinte estrutura da tabela após a aplicação de f :

n bits (fonte)	m bits (argumentos)	n bits (resultado)	m bits (resíduo)
0	X	Y	X
1	X	$Y + 1$	X
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2^n - 1$	X	$Y + 2^n - 1$	X

Para cada possível valor da fonte $a \in B^r$, $f(a, X) = (Y + a, X)$. Notar que $Y + a$ significa que para cada linha do bloco Y será somado a , com a soma realizada módulo 2^n . Como, em nosso exemplo, $n = 1$, se a é ímpar o bit é invertido.

c	x_1	x_2	y	g_1	g_2
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1

Por construção, cada lado da tabela resultante contém cada elemento de B^{m+n} uma única vez, o que torna a função inversível. E como vimos com XOR/FAN-OUT, outras funções podem aparecer quando a fonte assume valor que não zero. Em nosso exemplo atual, a função NAND é obtida para $c = 1$.

c	x_1	x_2	y	g_1	g_2
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1

- por fim, a função $v : B^n \times B^{r+m-n}$, que filtra os resíduos, extraindo apenas os bits de interesse.

Em resumo, as transformações são: $x \xrightarrow{\mu} (0^r, x) \xrightarrow{f} (\phi(x), \text{lixo}) \xrightarrow{v} \phi(x)$.

O número mínimo de bits extra para a reversibilização de funções dependerá da quantidade de informação que é perdida. A fórmula exposta evidencia casos em que toda a informação é perdida, como na função AND. Uma função de dois argumentos na qual o resultado mantém um deles e descarta o outro, nenhum bit extra precisará ser criado como fonte, somente o argumento perdido deverá compor um resíduo ou lixo.

x_1	x_2	y	$g = x_2$
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Todas as tabelas verdade trabalhadas representam as principais primitivas reversíveis já pensadas (ver Figura 6). É então a partir da composição dessas primitivas para a criação de circuitos complexos que são realizados os estudos acerca da reversibilidade.

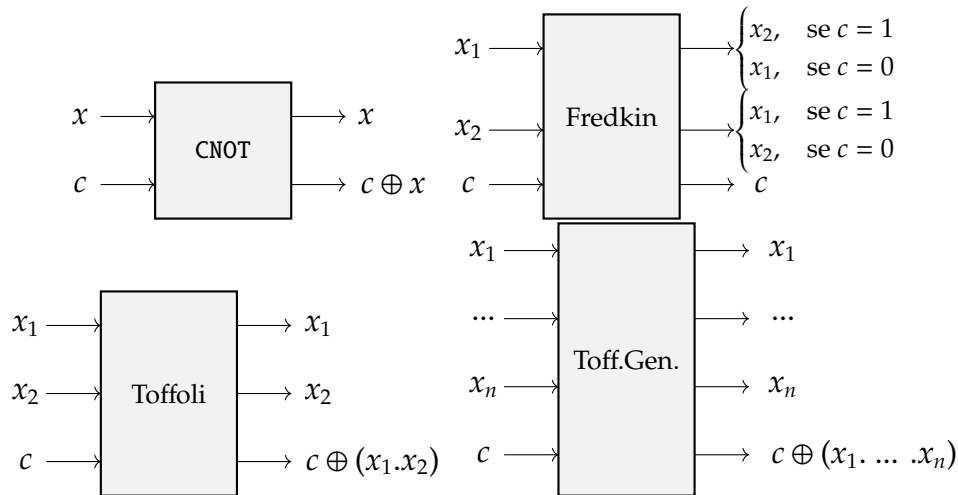


Figura 6: Portas lógicas reversíveis: CNOT (XOR/FAN-OUT), Fredkin (CSWAP), Toffoli (AND/NAND) e Toffoli generalizada.

A porta Fredkin foi proposta com propósito da implementação de uma lógica conservativa. Como a conservação de certas quantidades aditivas é também propriedade da física, Fredkin e Toffoli [24] visualizaram uma computação na qual zeros e uns atuam como entidades imutáveis e que apenas fluem entre os circuitos, mantendo portanto suas quantidades ao longo do tempo. A porta citada realiza o intercâmbio dos valores das entradas caso $c = 1$, senão x_1 e x_2 seguem com seus valores antigos. A lógica conservativa é reversível; não implica perda lógica de informação e, em princípio, permite implementações físicas nas quais a dissipação associada ao descarte de informação seja evitada. A reversibilidade lógica não depende, porém, de que seja conservativa [23].

A porta Toffoli é dita universal, pois consegue simular o funcionamento das portas AND, NOT e FAN-OUT, que perfazem um conjunto mínimo necessário para a construção de circuitos com o mesmo poder computacional de uma MT. A simulação é obtida através da fixação de determinados valores da entrada, como visto na Figura 7.

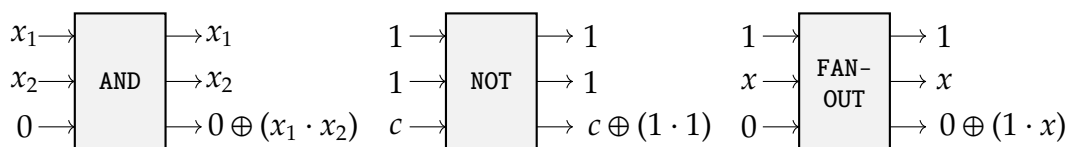


Figura 7: Toffoli como porta lógica universal

3.5 Bijetivação de problemas: um exemplo

Escolhemos a solução algorítmica apresentada por Glück e Yokoyama [26] por se tratar de um bom exemplo de reversibilidade condicional. Os autores visam mostrar que é preciso não desistir tão facilmente de uma função naturalmente irreversível, não recuando diante da tarefa de descobrir as condições nas quais esta pode ser reversibilizada, como visto na seção anterior.

Definição. O problema de correspondência de vetores de caracteres (*strings*), conhecido como *string-matching*, encontra aplicação em diversas áreas, a genômica sendo uma delas. Trata-se de encontrar as ocorrências de uma expressão ou padrão em um texto maior.

O padrão P e o texto T possuem tamanho, respectivamente, $m + 1$ e $n + 1$, com terminações $P[m] \neq T[n]$. O alfabeto Σ é composto de números inteiros, com $P[m] \notin \Sigma$. O algoritmo mais simples, dito ingênuo, é a aquele que compara P a cada grupo de mesmo tamanho no texto T . Ele compara $P[0]...P[m]$ com $T[0]...T[m]$, posição a posição,

enquanto os caracteres forem iguais ($P[i] = T[i]$) e enquanto não alcançado o final de P . Se todos os caracteres forem iguais, o índice de T referente ao início da comparação é uma ocorrência de P em T e então o valor é salvo. O procedimento é repetido para $T[1]$ até $T[n - m + 1]$ (Figura 8).

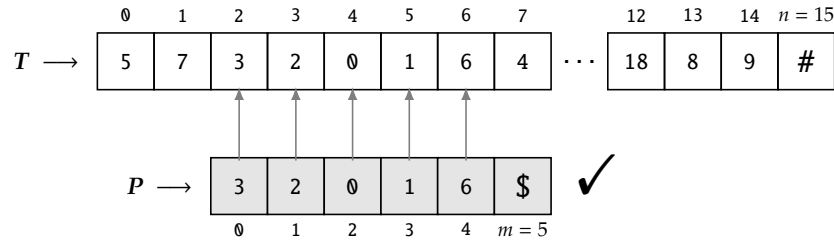


Figura 8: Exemplo de iteração no algoritmo ingênuo: o padrão P é comparado com a janela de T que se inicia na posição 2.

O algoritmo Rabin-Karp [10] aumenta a eficiência do algoritmo ingênuo, mesmo que não melhore sua complexidade de tempo no pior caso. Para que a cada vez não seja necessária a consulta de todas as posições de T e de P , é acrescentada uma função $hash : m \rightarrow 1$, que tem por objetivo mapear informação contida em m caracteres em um número que caiba no tamanho da palavra da máquina. Deve ser calculado o $hash\ p$ para o padrão P e para cada posição $T[s]$ de T . Sendo d o tamanho do alfabeto utilizado:

$$p = (P[0] \cdot d^{m-1} + P[1] \cdot d^{m-2} + \dots + P[m-1]) \bmod q$$

$$t_s = (T[s] \cdot d^{m-1} + T[s+1] \cdot d^{m-2} + \dots + T[s+m-1]) \bmod q$$

Como a função $hash$ realiza a compressão dos valores utilizando a operação que obtém o resto de uma divisão entre termos (**mod**), a comparação posição a posição será realizada somente quando $p = t_s$. A irreversibilidade causada pela compressão $m \rightarrow 1$ pode ser então contornada uma vez que P e T são mantidos.

Para que o cálculo dos polinômios seja mais eficiente, é utilizada a regra de Horner, que simplifica t_s para:

$$t_s = (\dots((T[s] \cdot d + T[s+1]) \cdot d + T[s+2]) \cdot d + \dots + T[s+m-2]) \cdot d + T[s+m-1] \bmod q$$

Por exemplo, se $m = 4$ e $d = 4$ e $q = 23$, e a sequência a partir de algum índice s é $\{3, 2, 0, 1\}$, então:

$$\begin{aligned}
t_s &= (3d^3 + 2d^2 + 0d + 1) \bmod q = \\
t_s &= (((3d + 2) \cdot d + 0) \cdot d + 1) \bmod q = \\
t_s &= (((3 \cdot 4 + 2) \cdot 4 + 0) \cdot 4 + 1) \bmod 23 = 225 \bmod 23 = 18
\end{aligned}$$

Adiantamos que o valor de q deve ser escolhido como o maior primo de modo que dq caiba em uma palavra da máquina.

Para que a proposta realmente aumente a eficiência do algoritmo, o cálculo de t_{s+1} deve aproveitar o valor de t_s , retirando-se a parcela devida a $T[s]$ e adicionando-se a contribuição de $T[s + m]$ após o restante ser multiplicado pela base d . Então, a fórmula recursiva é:

$$t_{s+1} = ((t_s - T[s] \cdot d^{m-1}) \cdot d + T[s + m]) \bmod q$$

A atualização de t_{s+1} passa a ser realizada em $\Theta(1)$ bastando para isso que d^{m-1} seja calculado de antemão.

A questão central é a da bijetivação da função de atualização de t , porque a utilização da operação aritmética de resto da divisão (**mod**) geralmente não é reversível. A função de atualização pode ser reescrita como a combinação de 3 operações aritméticas modulares $+_q$, $-_q$ e $*_q$, que os autores assumem estarem disponíveis: $+_q$, $-_q$ e $*_q$:

$$\begin{aligned}
t &= (t + T[s]) \bmod q && \longrightarrow t +_q T[s] \\
t &= (t \cdot d) \bmod q && \longrightarrow t *_q d \\
t &= (t - T[s + m]) \bmod q && \longrightarrow t -_q T[s + m]
\end{aligned}$$

De acordo com a definição de Copenhagen, uma função é bijetiva se é resultado da combinação de funções bijetivas. Os autores mostram que:

- $x +_q y$ e $x -_q y$ são bijetivas como funções de x , para um mesmo y constante e se $0 \leq x, y < q$; e
- $x *_q d$ é bijetiva se $0 \leq x < q$, $1 \leq d < q$, e d e q são coprimos: $\text{mdc}(d, q) = 1$.

Vale complementar que:

- $x -_q y$ é a inversa de $x +_q y$ e vice-versa. A aritmética modular forma um grupo cíclico sob adição, significando que cada número y tem um inverso aditivo único tal que $(y + (-y)) \bmod q = 0$. E como q é primo, o conjunto $\{0, 1, \dots, q-1\}$ cobre todos os resíduos possíveis sem ambiguidade. Ex:

$$x = 3, y = 5, q = 7$$

$$x = (3 + 5) \bmod 7 = 8 \bmod 7 = 1$$

$$x = (1 - 5) \bmod 7 = -4 \bmod 7 = 3 \text{ (sempre positivo)}^5$$

- Ao contrário da adição e da subtração, a multiplicação $x *_q d$ é sua própria operação inversa, que se utiliza, entretanto, não de d , mas de seu inverso multiplicativo (exclusivo para para mod q) d^{-1} tal que $(x *_q d)_q d^{-1} = x$. Tal inverso é calculado utilizando-se o algoritmo de Euclides estendido [10]. O algoritmo acha os coeficientes x e y na fórmula $xd + yq = 1$, fornecidos d e q . O cálculo é iterativo, iniciando com $x = 1$ e $y = 0$, e a cada iteração são promovidas as seguintes transformações:

$$\begin{cases} d' \leftarrow q \\ q' \leftarrow a \bmod q = d - \left\lfloor \frac{d}{q} \right\rfloor \cdot q \\ x' \leftarrow y \\ y' \leftarrow x - \left\lfloor \frac{d}{q} \right\rfloor \cdot y \end{cases}$$

As iterações continuam enquanto $a \neq 0$. Ao final, o inverso se encontrará em x . Se este for negativo, q deve ser adicionado para que então assuma valor positivo. Ex:

Iteração	d	q	Quociente	x	y
0	4	13	–	1	0
1	13	4	3	0	1
2	4	1	4	1	-3
3	1	0	–	-3	13
4	0				

Verificação:

$$d = 4, q = 13 \text{ e } \varphi = 7$$

$$\varphi' = (\varphi \cdot 4) \bmod 13 \rightarrow \varphi' = (7 \cdot 4) \bmod 13 = 2$$

⁵Aos programadores C/C++: o operador % não calcula corretamente quando há argumentos negativos. A expressão $a \bmod b$ deve ser substituída por $a - \text{int}(a/b) \cdot b$, sabendo-se que a função `int()` arredonda para baixo.

Se $d^{-1} = -3 \bmod 13 = \underline{10}$, então

$$\varphi = (\varphi' \cdot d^{-1}) \bmod 13 = (2 \cdot 10) \bmod 13 = 20 \bmod 13 = \boxed{7} \checkmark$$

Embora a função resto da divisão inteira, considerada isoladamente como aplicação de $\mathbb{Z}$ em $\mathbb{Z}_m$, não seja bijetiva, sua utilização no algoritmo ocorre como parte de transformações aritméticas definidas inteiramente no anel finito $\mathbb{Z}_m$, nas quais cada operação é uma permutação do espaço de estados. A reversibilidade decorre, portanto, não da bijetivação da função resto em si, mas da estrutura bijetiva das transformações modulares empregadas.

4 O encontro da teoria com a realidade

4.1 Computação energeticamente eficiente

A promessa de ser energeticamente eficiente é o que faz com que a computação reversível seja sedutora para a engenharia, não sua teoria. Ainda não foram implementadas portas totalmente reversíveis, cuja dificuldade na construção é conhecida [20]. Mas, como a tecnologia é sempre pressionada a avançar e não espera, podemos nos perguntar: a implementação de uma computação energeticamente eficiente pode prescindir da reversibilidade lógica dos circuitos digitais?

A resposta depende de uma análise mais precisa da conclusão primeva de Landauer, que ficou conhecida pela fórmula "a irreversibilidade lógica implica na irreversibilidade termodinâmica":

Considere um conjunto estatístico de bits em equilíbrio térmico. Se a eles for atribuído o valor UM, o número de estados cobertos no conjunto terá sido cortado pela metade. A entropia então terá sido reduzida por $k \log_e 2 = 0.6931 k$ por bit. A entropia de um sistema fechado, i.e., um computador com suas baterias, não pode diminuir; então essa entropia deve aparecer em outro lugar como efeito de calor, suprimindo $0.6931 kT$ por bit restaurado ao entorno. Esse é, óbvio, um mínimo efeito de calor, e nosso método de raciocínio não garante que esse mínimo seja alcançável na prática [39].

Ou seja, Landauer disse que a perda de informação leva a uma redução de estados lógicos distinguíveis, e que essa redução na entropia do sistema informacional da máquina deve ser compensada, de acordo com a 2ª lei da termodinâmica, por um aumento de igual valor na entropia das cercanias, o que implica na necessidade da dissipação, em forma de calor, da energia envolvida na informação perdida.

Lairez [38] não invalida Landauer, mas limita o alcance da sua afirmação, questionando a necessidade de dissipação final presente em sua conclusão. Propõe um modelo

no qual um sistema físico é conduzido, por meio de um processo quase-estático e controlado, a um estado final que corresponde logicamente ao bit 0. Os dispositivos assim construídos não realizam as transições bruscas de estados encontradas na computação tradicional e que levam à dissipação. Em vez disso, a operação é representada por ondas trapezoidais (Figura 9), nas quais as rampas pouco inclinadas indicam a transição mais suave das voltagens que representam os estados possíveis para o bit. Desta forma, a energia pode ser capturada e reutilizada¹, em um processo que envolve a atuação de dispositivos conhecidos como ressonadores.

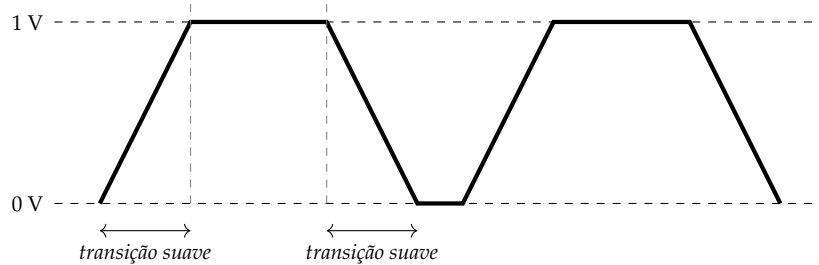


Figura 9: Onda trapezoidal, indicando a transição suave de voltagem que permite o aproveitamento da energia.

O quadro que descrevemos exemplifica a computação conhecida como *energeticamente eficiente* (*energy-efficient computation*) ou *adiabática*, na qual a recuperação de energia tem como limite assintótico o da reversibilidade total. A adiabaticidade tem uma face dupla: é certamente umas das tecnologias utilizadas nas tentativas de obtenção de portas totalmente reversíveis, mas também pode ser aplicada independentemente da adoção da reversibilidade lógica para reaproveitamento da energia utilizada na computação.

Então, a reversibilidade lógica garante que não haja dissipação por perda de informação no limite ideal, mas é possível também que a energia que estaria sentenciada à dissipação pela irreversibilidade seja reaproveitada no sistema não informacional da máquina.

Ao lado da computação logicamente reversível e da adiabática, os autores ainda citam a realimentação da fonte de energia a partir da recuperação da energia já dissipada como uma terceira técnica.

¹Para que possa ser reaproveitada, a energia capturada deve ser "pastoreada" para um armazenamento seguro (*safe storage facility*) ou seguir "ocupada", ou seja, têm-se que redirecionar para um outro trabalho a energia que acabou de realizar uma operação – senão ela mesma se ocupará de se dissipar [20].

4.2 Computador completamente reversível?

A adoção do paradigma reversível coloca no horizonte a possibilidade da computação isenta da perda de informação e, portanto, com dissipação zero de energia. Mas é possível, na prática, a construção de um computador completamente reversível? Ou quanto de um computador reversível pode ser reversível na teoria e na prática?

Para Lukac et al. [42], a resposta à primeira pergunta é não. Tal negativa é a conclusão de uma análise que se inicia pela assunção de que um computador completamente reversível é um sistema fechado (*closed reversible computer*), que deve ser conservativo e ideal com relação à energia envolvida exclusivamente no subsistema informacional e, por conseguinte, não pode ser operado desde fora. Mas essa figura é expulsa do paraíso quando imersa no mundo real irreversível com o qual deve interagir. Na prática, todo sistema é aberto em algum nível.

Nesse cenário, os fatores de dissipação são:

Transformação de entradas irreversíveis em entradas reversíveis. Toda transformação de estados contínuos do mundo físico em representações digitais (p.ex., o pressionar de uma tecla em um teclado) é uma função muitos-para-um, o que significa que opera uma compressão de estados que se traduz em irreversibilidade lógica.

Transformação de saídas reversíveis em saídas irreversíveis. A mudança de estado de um dispositivo físico de saída (p.ex., a cor de um led) se dá com o apagamento do estado anterior e consequente perda de informação.

Controle irreversível. Por controle entende-se os mecanismos de sincronização e administração do acionamento de chaves e escolhas de caminhos lógicos, que reconfigura fisicamente o circuito. O controle envolve sinal e comutação (*switching*)², e então gerará entropia fora do sistema informacional se o sinal (p.ex., o do *program counter* – PC) vier de uma fonte irreversível ou se a comutação não for fisicamente reversível. O sistema do relógio (*clock*) da máquina, que envolve a geração do sinal, sua distribuição e elementos acionados, pode chegar a consumir 40%-50% de toda a potência do microprocessador [30]. Pode ser idealizada uma forma de controle na qual as chaves são acionadas por circuitos reversíveis, o que significaria dizer que o controle seria internalizado pela máquina reversível. É conceitualmente possível confinar a irreversibilidade lógica e a dissipação de energia ao subsistema de *clock* e controle, enquanto

²Comutação é o processo físico de transição entre estados lógicos de um circuito, o qual envolve o carregamento e descarregamento de capacitâncias e o fluxo transitório de corrente [54].

o núcleo computacional permanece lógica e fisicamente reversível. Em implementações síncronas práticas, no entanto, o *clock* é inevitavelmente acoplado aos elementos que armazenam o estado, fazendo com que a dissipação se propague além do próprio gerador de *clock*.

O custo físico da computação em si. Dado que os 0 e 1 lógicos são mapeados por níveis físicos bem definidos de energia, pode parecer que a conservatividade energética implica na conservatividade lógica e vice-versa, o que não é correto. Tal conclusão não é válida universalmente, mas é dependente da tecnologia atual de representação física dos bits lógicos. No capítulo anterior vimos que a conservatividade lógica³ não é requisito para a reversibilidade, mas, se estamos contabilizando energia, o que pode acontecer é um sistema logicamente reversível não ser energeticamente conservativo. Em virtude desse fato e por definição, não teremos um sistema fechado, pois o somatório de sua energia oscilará para mais ou para menos dependendo da configuração lógica total assumida em cada momento pela máquina. Essa variação na energia não terá, no entanto, a ver com perda (ou ganho) de informação.

A dissipação também ocorre como resultado da comutação e da dissipação térmica resistiva de transmissão que ocorre através do grande número de interconexões presentes nos circuitos.

4.3 O fim da computação reversível?

Se a promessa da computação reversível levou, em um primeiro momento, à expectativa da construção de um computador completamente reversível e sem dissipação, torna-se necessário reconhecer os limites dessa visão. O que inicialmente se apresentava como um projeto tecnológico realizável deve ser compreendido, à luz da análise física e arquitetural, como um ideal teórico. Isso implica que a computação reversível deva ser descartada, ou que tenha perdido sua relevância?

A resposta é negativa.

A computação reversível permanece matematicamente bem definida, logicamente rigorosa e algoritmicamente fértil. Ela demonstra a possibilidade da inversão de computações, da eliminação de estados intermediários e da redução de espaço auxiliar necessário, tornando explícito o custo estrutural da reversão. Assim, sua contribuição é fundamental para a compreensão dos limites inferiores da dissipação associada à

³Definida como a manutenção da quantidade de estados lógicos 1 por toda a computação.

perda de informação.

Do ponto de vista prático, a reversibilidade lógica continua sendo um dos fatores possíveis de redução do consumo energético total. Em qualquer escala considerada — seja no nível global do circuito, de seus módulos ou de portas individuais — a reversibilidade lógica pode coexistir com fontes inevitáveis de dissipação, bem como com técnicas adiabáticas de recuperação de energia. Ela não elimina essas perdas, mas reduz a parcela associada à irreversibilidade informacional.

Caso perdas resistivas venham a ser substancialmente mitigadas, por exemplo, pela eventual descoberta de materiais supercondutores à temperatura ambiente, a economia de energia decorrente da adoção da reversibilidade lógica tenderá a se tornar mais significativa. Ainda assim, permanecerão presentes dissipações associadas à interação com o ambiente, ao controle e às operações de entrada e saída, reafirmando o caráter assintótico, não absoluto, desse limite.

4.4 Análise de alguns representantes

4.4.1 Pendulum

Pendulum [69, 68, 70, 23] foi um dos primeiros projetos completos de arquitetura e RTL (Register Transfer Level) de um processador reversível e funcionalmente testado, uma arquitetura reversível totalmente simulada, e que permanece como referência conceitual. O programa divide a mesma memória com os dados e possui 32 registradores de propósito geral do tipo inteiro com sinal, r_0 a r_{31} . Por convenção, r_0 contém sempre o valor 0, r_1 é o ponteiro para a pilha de chamada de funções e r_2 guarda o deslocamento embutido nas chamadas de funções. Seu conjunto de instruções, PISA (ver seção 6.2.1), acabou por se tornar um padrão utilizado e atualizado por estudos posteriores (ver 6.2.4). Younis1993, Younis [73] mostraram que era possível fabricar um circuito reversível; Vieri mostrou como seria um computador reversível. Posteriormente, Frank e Knight Jr [23] desenvolveram uma estrutura teórica geral que unificou os níveis lógico, físico e arquitetônico para a computação reversível.

A arquitetura possui características decisivas que asseguram a reversibilidade lógica e física:

Bit de direção (dir). Controla o incremento ou decremento do PC. É iniciado com 1, significando que o passo do PC é por incremento. Se o bit é invertido para -1, o PC passa

a ser decrementado. Essa foi a solução para tornar o clock é totalmente reversível, pois o PC faz parte do estado da máquina.

Simetria. Como o conjunto de instruções é lógica e fisicamente simétrico, a execução de uma adição ou de uma subtração é decidida pelo valor do DIR. Assim, cada transformação de estado é bijetiva e sem perda de informação.

Acesso à memória por troca (vs. atribuição destrutiva). Da mesma forma, o acesso à memória é executado através da instrução `EXCH  $r_d, r_s$` (*exchange*) que realiza a permuta dos valores do registrador r_d com o de uma posição de memória apontada por outro registrador, r_s , de forma atômica. Fisicamente, os dois valores são levados a trânsito e recepcionados em seus destinos, agora zerados, de modo que as atribuições assim realizadas não incorrem em perda de informação. A instrução é sua própria inversa.

Estados controlados. As instruções somente alteram registradores e posições de memória, de modo que não há sinalizadores (de *overflow etc.*) que possam causar efeitos colaterais e irreversibilidade.

Manutenção da reversibilidade do PC nos saltos. As atualizações não incrementais do PC implicam em perda de informação. A solução para possibilitar a reversão dos saltos foi a criação de um registrador especial, o BR (*branch register*), que participa da memorização e alteração do PC nesses procedimentos. O raio-X do salto incondicional pode ser visualizado na Fig. 12 do capítulo 6.

4.4.2 Máquina de Demaine et al. [12]

Em *Energy-efficient algorithms* [12] encontramos a definição de uma máquina teórica *semirreversível*, ou seja, uma que, ao contrário de Pendulum, permite a presença da irreversibilidade, em todos os níveis, desde as portas lógicas até o conjunto de instruções. Os autores não pretendem propor uma arquitetura fisicamente reversível completa, eliminar a dissipação do *clock* ou descrever um chip realizável, mas o de reconhecer a reversibilidade lógica como conceito chave para a eficiência energética e permitir a contabilidade da execução de operações irreversíveis ou de bits destruídos por um algoritmo através do que chamou de *complexidade de energia*. Assim, estabelecem um modelo algorítmico no qual o custo energético contabiliza exclusivamente as operações logicamente irreversíveis.

A reversibilidade implica na utilização de informações extras que permitem transformar as operações irreversíveis em reversíveis. O crescimento dessa necessidade de

espaço pode se tornar impeditivo para algumas aplicações, como sistemas embarcados. A ideia dos autores foi a de tornar possível a troca da reversibilidade total por menos complexidade de espaço através da permissão para a execução de instruções que destroem informação, ao gosto da situação. Assim, surge a figura do algoritmo semirreversível.

Nessa máquina teórica, o PC avança monotonicamente, o que já constitui uma fonte de irreversibilidade se esse registrador especial integrar o estado lógico da máquina, pois que implica perda de informação a cada transição, acarretando custo energético proporcional. Essa característica impõe uma diferença estrutural importante em relação à máquina Pendulum, especialmente no que diz respeito à programação e à possibilidade de reversão da execução, aspecto que será analisado no capítulo 6.

4.4.3 Ice River

A *startup* Vaire afirma que já conseguiu alcançar um reaproveitamento de 50% de energia com seu núcleo computacional reversível parcialmente genérico, batizado de Ice River [35]. A empresa, que aposta fortemente na computação com energia próxima a zero (*near-zero energy computing*), nos permite saber um pouco através de seus white papers [66] e [65].

Primeiramente, foi invertido o paradigma que privilegia o *hardware* (*hw*) em detrimento do *software* (*sw*). Tradicionalmente é a máquina que determina o que poderá ser feito nos níveis superiores, ou seja, $hw \rightarrow \text{sistema operacional} \rightarrow \text{compiladores} \rightarrow sw$ de aplicação, mas a Vaire propõe a utilização do paradigma *software-first*, o que afeta não só a engenharia de *hw* mas a filosofia de design computacional de forma geral. Se o que vinga é que: aplicações $\rightarrow sw \rightarrow hw$ sob demanda, então o desenvolvimento começa pelas demandas de *sw* e o *hw* é customizado para atender de forma mais eficiente possível a essas demandas a partir da disponibilização de unidades especializadas, como multiplicadores matriciais reversíveis, buffers otimizados para tipos de dados específicos, etc.

O Ice River é voltado para cargas de trabalho com alta demanda aritmética – como IA, operações matriciais, simulações científicas –, e pensado para fazer parte de um sistema modular e escalável, sob controle de um *sw* que conhece a arquitetura reversível e que o ativa seletivamente quando identifica a oportunidade de haver ganhos termodinâmicos. A integração com outros dispositivos se dará via controle

orquestrado, e os módulos podem ser empilhados em *racks* de alta densidade ou embutidos em arquiteturas híbridas.

Através dos documentos entende-se que se trata da utilização conjunta da reversibilidade lógica ao lado de ressonadores para recuperação de energia em virtude da presença de irreversibilidades.

5 Interlúdio: a inversão de programas

A ideia da inversão de programas surgiu independentemente das pesquisas sobre os limites termodinâmicos da computação. Para este trabalho, distinguimos inversibilidade e reversibilidade, sendo a segunda um caso especial da primeira, caracterizado pela restrição a comandos que destroem informação. Por isso, não surpreende que a computação reversível se beneficie das técnicas de inversão de programas.

Dijkstra, em seu manuscrito de 1979-1980 intitulado *Program Inversion* [14], apresentou técnicas e condições para obter um programa P^{-1} que calcula o inverso de um original P . A motivação foi a constatação de que alguns problemas são mais facilmente resolvidos se atacados ao contrário: dado um programa P que recebe E como entrada e produz S como saída para um problema mais simples, obter P^{-1} permite recuperar E a partir de S , resolvendo assim um problema mais difícil. Dijkstra ressaltou que o sucesso do seu exemplo dependia de ser “um programa determinístico no qual nenhuma informação é destruída”.

As técnicas foram sistematizadas por Gries [29], que reconheceu bijetividade como condição essencial da inversão e propôs fórmulas para comandos isolados, blocos, condicionais e estruturas de repetição. O método se apoia em pré-condições e pós-condições, organizadas como trípulas da lógica de Floyd-Hoare [19, 33], que devem ser verificadas antes e depois da execução de cada comando ou bloco.

Alguns comandos simples não são inversíveis isoladamente, como a atribuição pura: $\boxed{\{x = 3\} \ x \leftarrow 1 \ \{x = 1\}}$.

Nesse caso, não há como determinar o valor anterior de x apenas pela execução do comando. Em termos de reversibilidade física, ocorre perda de informação, com dissipação de energia. Entretanto, em inversão de programas, o valor inicial é conhecido pelo programador e pode ser restaurado utilizando o truque da inversão da pré-condição: $\boxed{\{x = 1\} \ x \leftarrow 3 \ \{x = 3\}}$.

Já atribuições reversíveis — nas quais a variável aparece nos dois lados da expressão,

como incrementos e decrementos ($x \leftarrow x + 1$, $x \leftarrow x - 1$) — possuem inversa direta e dispensam truques adicionais.

De forma geral, a inversão de uma sequência de comandos ou blocos C_i segue a regra:

$$C_1; C_2; \dots; C_n \xrightarrow{\text{inversão}} C_n^{-1}; \dots; C_2^{-1}; C_1^{-1}$$

Com a ressalva de que comandos destrutivos devem ser tratados com pré-condições invertidas, como mostrado em:

$$x = k_1; C \{x = k_2\} \xrightarrow{\text{inversão}} x = k_2; C^{-1} \{x = k_1\}$$

Para variáveis booleanas B e R , com valores verdadeiro (B, R) ou falso ($\neg B, \neg R$), a inversão de condicionais é dada por:

$$\begin{array}{ccc} \begin{array}{l} \{B \text{ ou } \neg B\} \\ \text{se } B \text{ então} \\ \quad C_1 \{R\} \\ \text{senão} \\ \quad C_2 \{\neg R\} \\ \text{fim-se } \{R \text{ ou } \neg R\} \end{array} & \xrightarrow{\text{inversão}} & \begin{array}{l} \{\neg R \text{ ou } R\} \\ \text{se } \neg R \text{ então} \\ \quad C_2^{-1} \{\neg B\} \\ \text{senão} \\ \quad C_1^{-1} \{B\} \\ \text{fim-se } \{\neg B \text{ ou } B\} \end{array} \end{array}$$

A inversão mantém as relações: $\{B\} C_1 \{R\}$ e $\{\neg B\} C_2 \{\neg R\}$ em $\{R\} C_1^{-1} \{B\}$ e $\{\neg R\} C_2^{-1} \{\neg B\}$.

Para loops enquanto, a inversão é:

$$\begin{array}{ccc} \begin{array}{l} \{R\} \\ \text{enquanto } B \text{ faça} \\ \quad C \{\neg R\} \\ \text{fim-enquanto } \{\neg B\} \end{array} & \xrightarrow{\text{inversão}} & \begin{array}{l} \{\neg B\} \\ \text{enquanto } \neg R \text{ faça} \\ \quad C^{-1} \{B\} \\ \text{fim-enquanto } \{R\} \end{array} \end{array}$$

Exemplo com $R : \{x = 10\}$, $\neg R : \{x \neq 10\}$, $B : \{x \neq 0\}$ e $\neg B : \{x = 0\}$:

$$\begin{array}{ccc} \begin{array}{l} \{x = 10\} \\ \text{enquanto } x \neq 0 \text{ faça} \\ \quad C \\ \quad x \leftarrow x - 1 \{x \neq 10\} \\ \text{fim-enquanto } \{x = 0\} \end{array} & \xrightarrow{\text{inversão}} & \begin{array}{l} \{x = 0\} \\ \text{enquanto } x \neq 10 \text{ faça} \\ \quad x \leftarrow x + 1 \\ \quad C^{-1} \{x \neq 0\} \\ \text{fim-enquanto } \{x = 10\} \end{array} \end{array}$$

5.1 C inversível

A linguagem C [37] possui construções irreversíveis, mas técnicas de transformação *source-to-source* permitem gerar P_+ e P_+^{-1} [51, 52]. O método utiliza uma pilha para salvar variáveis que seriam destruídas e registros de caminhos de fluxo, sem alterar a semântica do programa.

As funções auxiliares utilizadas são:

SAVE(x)	empilha <i>lvalue</i> x
RESTORE(x)	recupera <i>lvalue</i> x
SAVE_BITS(x, n)	empilha n bits ‘mais baixos’ de <i>lvalue</i> x
RESTORE_BITS(x, n)	recupera n bits ‘mais baixos’ de <i>lvalue</i> x

As etapas que aplicadas em sequência sobre o código são as de normalização, transformação e otimização. O resumo das operações aplicadas são resumidas a seguir.

Normalização. Todas as formas de quebra de fluxo (*case*, *break*, *return*) são transformadas em salto *goto* e as declarações de variáveis são separadas de suas inicializações. Os efeitos colaterais são removidos e os saltos devem verificar a necessidade de salvamento de variáveis de escopo.

Construção	Original	Normalizada
Declarações: separar das atribuições	<code>int x = 10, y = 20;</code>	<code>int x, y; [x = 10; y = 20;]</code>
Efeitos colaterais (EC): eliminar	<code>y = (++x) + (z++);</code>	<code>++x; [y = x + z;] z++;</code>
EC em condicionais: eliminar	<code>while(x++ > 0) {...}</code>	<code>while([x > 0]) {...; [x++;] }</code>
Chamadas: como único <i>rvalue</i>	<code>x = y + f();</code>	<code>[T t;] [t = f();] x = y + t</code>
Expressões: aumentar precisão	<code>float x, y, z; x = y + z</code>	<code>[double x, y, z;] x = y + z</code>
for: transformado em while	<code>for(ini(x); test(x); inc(x)) s;</code>	<code>[ini(x);] while([test(x)]) { s; [inc(x);] }</code>
continue: salto para o fim do bloco	<code>{...; continue; ...}</code>	<code>{...; [goto label;] ...; [label:] }</code>
Múltiplos return: saltos para único	<code>{...; return x; ...; return y; ...}</code>	<code>{ [T t;] ... [t = x; goto label;] ... [t = y; goto label;] ... [label: return t; }</code>
break: salto para fora do bloco	<code>{...; break; ...} s</code>	<code>{...; [goto label;] ...; [label:] s</code>

Construção	Original	Normalizada
switch: if com goto	<pre>switch(expr) { case c_1: s_1; break; case c_2: s_2; /* fall-thru */ default: s_d; break; }</pre>	<pre>{ <u>int</u> c; $c = expr$; if($c == c_1$) goto $label_1$; if($c == c_2$) goto $label_2$; goto $label_d$; <u>label₁</u>: s_1; goto end; <u>label₂</u>: s_2; /* fall-thru */ <u>label_d</u>: s_d; <u>end</u>: }</pre>
Salto de escopo: salvar as variáveis (deve ser a última normalização)	<pre>{ int x; ... {int y; ...; goto label; ...} ... } label:</pre>	<pre>{ int x; ... { int y; ...; <u>SAVE</u>(y); <u>SAVE</u>(x); goto label; ... } ... } label:</pre>

Transformação. São gerados os códigos referentes a P_+ e P_+^{-1} através da aplicação das transformações descritas abaixo.

Normalizada	Transformada	Inversa
Expressões: os operandos ++, += e /= têm como inversos --, -= e *=. Todos os outros são destrutivos e requerem o salvamento do valor do operando.		
$lvalue\ op\ rvalue;$	{ <u>SAVE</u> ($lvalue$); $lvalue\ op\ rvalue;$ }	{ <u>RESTORE</u> ($lvalue$); }
Chamada simples de funções: a inversão do comando se dá pela substituição do nome da função pelo de sua inversa.		
$f_1(...); \dots ; lvalue = f_2(...)$	$f_1(...); \dots ; lvalue = f_2(...)$	$lvalue = f_2^{-1}(...); \dots ; f_1^{-1}(...);$
Blocos de comando (escopo): as variáveis locais devem ser salvas antes da saída do escopo. A reversão deve restaurá-las antes da execução da inversão dos comandos em ordem inversa.		
<pre>{ int $x, y; s_1; s_2; s_3;$ }</pre>	<pre>{ int $x, y; s_1; s_2; s_3;$ <u>SAVE</u>(x); <u>SAVE</u>(y); }</pre>	<pre>{ int $x, y;$ <u>RESTORE</u>(y); <u>RESTORE</u>(x); $s_3^{-1}; s_2^{-1}; s_1^{-1};$ }</pre>
Salto: variável memoriza a origem do salto.		

Normalizada	Transformada	Inversa
<pre>{ s₁; goto label; ... s₂; goto label; ... s₃; ... label: s₄; }</pre>	<pre>{ s₁; SAVE(1); goto label; ... s₂; SAVE(2); goto label; ... s₃; SAVE(0); ... label: s₄; }</pre>	<pre>{ char c; s₄⁻¹; RESTORE(c); switch(c) { case 0: break; case 1: goto label₁; case 2: goto label₂; } s₃⁻¹; ... label₂: s₂⁻¹; ... label₁: s₁⁻¹; ... }</pre>
Condicionais: o resultado do teste é transformado em bit, testado e salvo ao final, para que a reversão possa saber qual bloco invertido de comandos executar.		
<pre>if(cond) s₁ else s₂</pre>	<pre>{ char c = cond; if(c) s₁ else s₂ SAVE_BITS(c, 1); }</pre>	<pre>{ char c; RESTORE_BITS(c, 1); if(c) s₁⁻¹ else s₂⁻¹ }</pre>
While: cria variável para contar a quantidade de repetições executadas.		
<pre>while(cond) s</pre>	<pre>{ int c; c = 0; while(cond) { c++; s; } SAVE(c); }</pre>	<pre>{ int c; RESTORE(c); while(c) { s⁻¹; c--; } }</pre>
Do-while: mesma técnica utilizada no while.		
<pre>do s while(cond);</pre>	<pre>{ int c; c = 0; do { c++; s; } while(cond); SAVE(c); }</pre>	<pre>{ int c; RESTORE(c); do { s⁻¹; c--; } while(c > 0); }</pre>
Switch: esta é a outra alternativa para quando o switch não foi normalizado. Variáveis memorizam a opção selecionada e detectam a ocorrência de <i>fall-thru</i> .		
<pre>switch(expr) { case v₁: s₁; break; case v₂: s₂; /* fall-thru */ default: s_d; }</pre>	<pre>{ char c, d; c = 0; d = 0; switch(expr) { case v₁: c = 1; d++; s₁; break; case v₂: c = 2; d++; s₂; default: c = 0; d++; s_d; } SAVE(c); SAVE(d); }</pre>	<pre>{ char c, d; RESTORE(d); RESTORE(c); switch(c) { case 0: s_d⁻¹; if(--d <= 0) break; case 2: s₂⁻¹; if(--d <= 0) break; case 1: s₁⁻¹; } }</pre>

Otimização. Esta fase tem como objetivo aumentar a eficiência temporal e espacial do programa. Listamos alguns casos de evitação de salvamento e restauro de valores:

- variáveis cujos valores têm como destino único o envio através de um dispositivo externo (impressão, comunicação *etc.*), por exemplo;
- os valores iniciais das variáveis, os chamados *don't care values*;
- os relativos a expressões que não se alteram em blocos de código, o que significa que o procedimento pode tratar o código com uma granularidade maior¹;
- situação em que é avaliado que somente alguns valores assumidos por uma variável em uma repetição são importantes no que o salvamento de valores prévios podem ser substituídos pelo de uma contagem.

5.2 Alguns paradigmas da inversão e suas aplicações

Uma das aplicações ou a variação mais famosa da inversão de programas é a facilidade de desfazer e de refazer comandos ou tarefas executadas, o que valeu até a universalização das teclas de atalho Ctrl-Z e Ctrl-X para essas funções. Originalmente utilizadas por editores de texto em microcomputadores, tal facilidade de permitir que a máquina recomponha estados anteriores tem sido implementada em diversos outros contextos tais como: sistemas tolerantes a falhas, banco de dados, linguagens e ambientes de programação. Leeman Jr [41] apresenta um extenso histórico dos trabalhos realizados em cada uma dessas áreas, e propõe uma estrutura para desenvolvimento (*framework*) com a finalidade de adicionar a facilidade de *undo* a linguagens de programação. Briggs [8] consegue desfazer comandos de atualização de placar eletrônico em jogos de cricket através de uma solução de inversão.

Apresentamos então três dos principais paradigmas da inversão de programas [52], além do computa-copia-reverte estudado em 3.1:

Desfazer-refazer-fazer (Undo-redo-do: URD). Utilizado em editores de texto e imagem, depuradores (*debuggers*) e modelos de computação reversível. A esquematização do paradigma é apresentado na Fig. 10.a. Uma ação ou um grupo de ações P de um programa que avançou ($D(P)$) podem ser desfeitas ($U(P)$) e refeitas ($R(P)$) repetidas vezes por escolha ($\circ$) do usuário, até produzir sua saída normal (E , resultante de $D(P)$) ou vazia ($\square$, se $R(P)$ for seu último comando executado).

¹Ver conceito de *proteção* na seção 6.2.3.

Avançar-reverter-confirmar (*Forward–reverse–commit*): utilizado nas situações nas quais um fragmento do programa executa otimamente antes mesmo das suas dependências de dados estarem satisfeitas completamente (*ahead of time*). Quando alguma dessas dependências é atualizada, a execução anterior ($F(P)$) é desfeita ($R(P)$) e o fragmento é reexecutado. O processo é repetido até que obtenha a garantia de que não ocorrerão mais atualizações, quando a execução então é confirmada ($C(P)$) – Fig. 10.b. Utilizado em simulações de eventos discretos paralelos e na execução especulativa em nível de hardware².

Iniciar-reverter-confirmar (*begin–rollback–commit*): modelo transacional utilizado em banco de dados para garantir a atomicidade da execução de um bloco de operações (uma transação T_i), i.e., ou uma sequência de operações é executada com sucesso, como se uma operação única fosse e então é confirmada ($C(T_i)$), ou possíveis execuções parciais ($R(T_i)$) são revertidas para manter a consistência dos dados armazenados ($\square_i$, resultado nulo). O modelo pode ser aprimorado para considerar cada operação como sendo uma subtransação (T_{i+1}) que, se não for confirmada, inicia reversão em cadeia que alcança o nível mais externo de transação – Fig. 10.c.

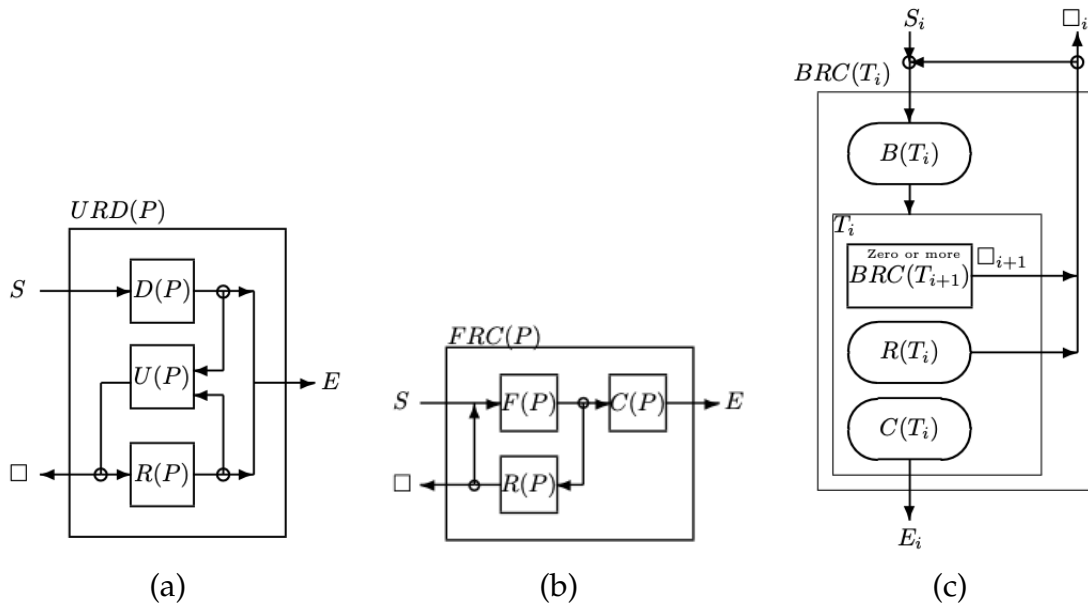


Figura 10: Paradigmas utilizáveis no contexto da inversão: (a) Desfazer-Refazer-Fazer; (b) Avançar-Reverter-Confirmar; e (c) Iniciar-Reverter-Confirmar.

²*On-chip speculative execution*: o chip “adivinha” quais instruções virão a seguir e as executa antes da confirmação, visando ganho em desempenho.

5.3 Checkpointing

Complementarmente, podemos categorizar diferentes tipos de implementação de inversão através da utilização de pontos de controle conhecidas como *checkpointing*. Os *checkpoints* nada mais são do que momentos da execução de um programa nos quais as informações necessárias à inversão são salvas. Listamos quatro técnicas [52], cada um tendo sua aplicabilidade conforme o caso.

Em lados opostos estão os *checkpointings total* e *periódico*. No primeiro, todo o estado da máquina é salvo a cada passo; no segundo, somente após certa quantidade de operações executadas. Neste caso, para restaurar o estado da máquina ao de uma operação X intermediária a dois *checkpoints*, deve ser utilizado o *checkpoint* C imediatamente anterior a X e serem reexecutadas as operações desde C até X .

O terceiro tipo é utilizado em programas cujas ações não alteram todo o estado da máquina, sendo necessário que cada momento sejam salvas apenas as variáveis que serão modificadas a curto ou médio prazo. Tal abordagem *incremental* lhe dá o nome. Essa abordagem é mais eficiente do que as duas primeiras somente se o número de variáveis efetivamente modificadas é muito menor do que o número total de variáveis potencialmente modificáveis em uma ação.

Finalmente, o *checkpointing diferencial*. Seja S_i o estado atual da máquina. Chamemos de Δ_i a diferença estrutural entre S_i e o *checkpoint* imediatamente anterior C_i ($\Delta_i = S_i - C_{i-1}$). O *checkpoint* C_i atual então é obtido a partir da soma do *checkpoint* anterior C_{i-1} com Δ_i ($C_i = C_{i-1} + \Delta_i$). A reconstrução de um estado S_i resulta da aplicação dos Δ acumulados:

$$S_n = C_0 + \sum_{i=1}^n \Delta_i$$

A utilização da abordagem incremental é apropriada quando o programa modifica pouco os estados e de forma esparsa.

Observa-se que todos os paradigmas apresentados obtêm reversão mediante armazenamento auxiliar de informação, o que os distingue da reversibilidade lógica estrita discutida nos capítulos anteriores.

6 Programação reversível

6.1 Duas estruturas: a língua de sogra e o tapete

No capítulo 4 apresentamos dois modelos teóricos. O primeiro, a máquina Pendulum, que leva a reversibilidade às últimas consequências, na qual o PC participa do estado da máquina. O segundo, a máquina semirreversível de Demaine et al. [12], na qual o *clock* avança monotonicamente. A diferença entre os modelos deixa sua marca na estrutura geral dos programas escritos e executados em cada um.

Em Pendulum, a execução do programa assemelha-se ao do brinquedo "língua de sogra"¹: se estende durante a fase direta e se retrai automaticamente durante a fase reversa, sem necessidade de codificação extra. Essa retração não resulta de uma leitura textual inversa, mas da aplicação sistemática das operações inversas segundo a dinâmica reversível do controle. Já no modelo de Demaine et al. [12], o programa comporta-se mais como um tapete: tudo o que faz é desenrolar. A reversão exige uma fase explicitamente codificada a partir das informações produzidas durante a execução direta.

As estruturas de controle de fluxo presentes nas linguagens de alto nível, tais como os condicionais, as repetições e as chamadas de funções, seguem essa regra. Em Pendulum (ver seção 6.4), são escritas como um bloco único, no qual o código que testa a condição de saída na fase direta é o mesmo que testa a condição de entrada na fase reversa, e vice-versa. No modelo de Demaine et al. [12], essas estruturas precisam ser explicitamente partidas em dois códigos, uma para cada fase da execução.

A separação explícita entre as fases direta e reversa confere maior flexibilidade à estrutura do programa, uma vez que a execução reversa não precisa reproduzir simetricamente o controle de fluxo da fase direta. As condições que governam a entrada nas estruturas de controle de fluxo na fase reversa podem diferir das condições avaliadas

¹Nos Estados Unidos é conhecido como *party horn* ou *party blower*.

na fase direta desde que sejam funções da informação preservada por esta última. Para que a reversibilidade lógica seja preservada, é necessário, nos condicionais, por exemplo, que a escolha de ramos na fase direta seja registrada explicitamente ou permaneça inferível de modo unívoco a partir do estado resultante, de forma que a composição das duas fases resulte na identidade sobre o conjunto de estados efetivamente alcançados pela execução direta.

6.2 Linguagens reversíveis

6.2.1 PISA

PISA (*Pendulum Instruction Set Architecture*), como dissemos na subseção 4.4.1, é o conjunto de instruções do projeto Pendulum. Os principais comandos da versão que consideramos [2, 23, 3] são os definidos na Fig. 11. O controle de fluxo reversível é realizado através da combinação de 3 registradores: PC (*program counter*), BR (*branch register*) e dir, que indica a direção do passo base do PC – vale 1 para a fase direta, e -1 na fase reversa, quando as instruções são interpretadas como suas inversas (adição como subtração etc.).

$prog$	$::= ((l :)^? i)^+$	(programa)
i	$::= \text{ADD } r_d, r_s \mid \text{SUB } r_d, r_s$	$(r_d \leftarrow r_d + r_s \mid r_d \leftarrow r_d - r_s)$
	$\mid \text{ADDI } r_d, c \mid \text{ADDI } r_d, -c$	$(r_d \leftarrow r_d + c \mid r_d \leftarrow r_d - c)$
	$\mid \text{ANDX } r_d, r_s, r_t \mid \text{ANDIX } r_d, r_s, c$	$(r_d \leftarrow r_s \text{ AND}_{bit} r_t \mid r_d \leftarrow r_s \text{ AND}_{bit} c)$
	$\mid \text{NORX } r_d, r_s, r_t$	$(r_d \leftarrow \neg(r_s \text{ OR}_{bit} r_t))$
	$\mid \text{NEG } r$	$(r \leftarrow \neg r)$
	$\mid \text{ORX } r_d, r_s, r_t \mid \text{ORIX } r_d, r_s, c$	$(r_d \leftarrow r_s \text{ OR}_{bit} r_t \mid r_d \leftarrow r_s \text{ OR}_{bit} c)$
	$\mid \text{RL } r, c \mid \text{RR } r, c$	$(r \leftarrow r \ll c \mid r \leftarrow r \gg c)$
	$\mid \text{RLV } r_a, r_b \mid \text{RRV } r_a, r_b$	$(r_a \leftarrow r_a \ll r_b \mid r_a \leftarrow r_a \gg r_b)$
	$\mid \text{SLLX } r_d, r_s, c \mid \text{SLRX } r_d, r_s, c$	$(r_d \leftarrow r_s \ll c \mid r_d \leftarrow r_s \gg c)$
	$\mid \text{SLLVX } r_d, r_s, r_t \mid \text{SRLVX } r_d, r_s, r_t$	$(r_d \leftarrow r_s \ll r_t \mid r_d \leftarrow r_s \gg r_t)$
	$\mid \text{SRAX } r_d, r_s, c \mid \text{SRAVX } r_d, r_s, r_t$	(=SLRX =SRLVX, mantendo o sinal)
	$\mid \text{XOR } r_d, r_s \mid \text{XORI } r, c$	$(r_d \leftarrow r_d \text{ XOR}_{bit} r_s \mid r \leftarrow r \text{ XOR}_{bit} c)$
	$\mid \text{BEQ } r_a, r_b, l \mid \text{BGEZ } r, l \mid \text{BGTZ } r, l$	$(\text{PC} \leftarrow l, \text{ se } r_a = r_b \mid r \geq 0 \mid r > 0)$
	$\mid \text{BLEZ } r, l \mid \text{BLTZ } r, l \mid \text{BNE } r_a, r_b, l$	$(\text{PC} \leftarrow l, \text{ se } r \leq 0 \mid r < 0 \mid r_a \neq r_b)$
	$\mid \text{BRA } l$	$(\text{PC} \leftarrow l)$
	$\mid \text{RBRA } l$	$(\langle \text{PC}, \text{BR} \rangle \leftrightarrow \langle l, \text{PC} \rangle)$
	$\mid \text{SWAPBR } r$	$(\text{BR} \leftrightarrow r)$
	$\mid \text{EXCH } r_a, r_b$	$(r_a \leftrightarrow \text{mem}(r_b))$
	$\mid \text{START} \mid \text{FINISH}$	(início e fim de programa)
	$\mid \text{DATA } c$	$(\text{mem}(\text{DATA}) \leftarrow c, \text{ inicialização de memória})$
c	$::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$	(numeral)

$prog \in \text{Programas}$
 $r \in \text{Registradores}$

$i \in \text{Instruções}$
 $l \in \text{Rótulos}$

Figura 11: PISA – *Pendulum Instruction Set Architecture* [69] [2] [32].

Saltos. Um comando de salto nada mais faz do que adicionar o valor de deslocamento (*offset*) contido em sua instrução, multiplicado por *dir*, ao PC. O modelo de língua de sogra faz com que os saltos sejam organizados dentro da filosofia conhecida como *paired branches* (ver Figura 12), segundo a qual um salto espera encontrar um outro comando de salto com *offset* inverso, no pouso, de modo que o BR seja zerado e o passo do programa retorne a 1 (ou -1), e de modo que, encontrando o segundo comando na fase reversa, o programa possa retornar ao ponto original do primeiro salto. A reversão de problemas mais complexos, porém, pode se utilizar de saltos que não zeram completamente o BR para obter seus resultados.

$$\begin{array}{l} \text{label}_1: \quad \text{BRA } \text{desloc} \quad ; \text{BR} \leftarrow \text{BR} + \text{desloc}; (\text{PC}, \text{BR}) \leftarrow (\text{PC} + \text{BR}, \text{PC}_{\text{anterior}}) \\ \text{desloc} \left\{ \begin{array}{l} \dots \\ \text{label}_2: \quad \text{BRA } -\text{desloc} \end{array} \right. \end{array}$$

Figura 12: Mecânica do salto incondicional reversível em PISA

Em resumo, o $\text{PC}_{\text{novo}} \leftarrow f(\text{PC}_{\text{anterior}}, \text{BR}, \text{dir})^2$ (ver Tabela 5). Os comandos de salto mais importantes são: BRA (incondicional), BEQ (salta se parâmetros possuem valores iguais) e BNE (se diferentes); RBRA acrescenta a inversão do registrador *dir*.

BR (1º)	Calculado sempre no salto; no pouso: BEQ, BNE: se true; BRA: sempre	$\text{dir} = 1 : \text{BR} \leftarrow \text{BR} + \text{desloc}$	$\text{dir} = -1 : \text{BR} \leftarrow \text{BR} - \text{desloc}$
PC (2º)		$\text{BR} = 0 : \text{PC} \leftarrow \text{PC} + \text{dir}$	$\text{BR} \neq 0 : \text{PC} \leftarrow \text{PC} + \text{BR} \cdot \text{dir}$
RBRA	Inverte <i>dir</i> (1 ou -1) As instruções passam a executar suas reversas	$\text{BR} = \text{BR} + \text{desloc} \cdot \text{dir}$	

Tabela 5: Quadro descritivo das possibilidades para atualização do PC em PISA, envolvendo os registradores BR e *dir*. A atualização do BR precede a do PC. A literatura consultada é ambígua sobre o momento da inversão de *dir* durante a execução da instrução RBRA.

A Fig. 13 apresenta como pequeno exemplo um programa de multiplicação de dois valores, *M* e *N*.

Os movimentos entre posições de memória são executados através de trocas atômicas de seus valores, o que evita qualquer perda de informação. Essa instrução, EXCH, é sua própria inversa.

²Embora pareça uma sobrescrição, trata-se de parte de uma transformação global bijetiva sobre o estado da máquina. De qualquer forma, nesse ponto: $\text{PC} + \text{BR} + (-\text{desloc}) = 0$.

```

main:
1:  ADDI R1,N
2:  ADDI R2,M
3:  BRA 3
4:  BRA 3
5:  BNE R4,R0,-1
6:  BEQ R4,R0,-3
7:  ADD R3,R1
8:  ADDI R2,-1
9:  ADDI R4,+1
10: BEQ R2,R0,2
11: BNE R2,R0,-7
12: BRA -2
13: SHOW R3
14: RBRA 0

```

Figura 13: Programa em PISA para cálculo de $N \times M$. Os registradores R1 e R2 são inicializados com os parâmetros da multiplicação. Na fase direta, R3 acumula as R2 adições de R1, onde R2 é decrementado e R4 é incrementado a cada iteração. O resultado R3 é mostrado (por duas vezes) e o passo do PC é invertido na linha 14. Na fase reversa, as operações das linhas 7 a 9, coração do *loop*, são invertidas, bem como as das linhas 1 e 2.

6.2.2 Janus

A primeira linguagem imperativa de alto nível totalmente reversível [43] foi criada inicialmente desvinculada de qualquer questão termodinâmica e como um exercício de semântica³. Solidificou sua posição após ser resgatada por Yokoyama e Glück [72] e modificada por Yokoyama, Axelsen e Glück [71]. Janus possui uma sintaxe simples (Fig 14), e todos os seus comandos e estruturas de controle de fluxo são bijetoras.

Janus admite exclusivamente o tipo inteiro, em sua forma simples, vetores de tamanho fixo e pilhas dinâmicas. As expressões binárias admitem operadores aritméticos, lógicos, relacionais e a nível de bit. Suas instruções básicas incluem atribuições aritméticas reversíveis: $x += y, x -= y$ etc.; trocas de valores entre variáveis: $x \Leftrightarrow y$; chamadas de procedimento com parâmetros passados por referência e chamadas inversas. Os métodos básicos sobre pilhas estão disponíveis, além de se poder acessar o topo sem removê-lo. Variáveis locais são introduzidas a partir da definição de um bloco de código como seu escopo e são removidas sob a condição de restauração do valor inicial. O símbolo $\wedge$ representa a operação *ou-exclusivo* (XOR).

O controle de fluxo é realizado pelos comandos mostrados na Fig. 15.

³Em carta enviada para apresentação do trabalho a ninguém menos do que Landauer, Chris Lutz confessou que a linguagem nasceu "por curiosidade sobre se um animal tão estranho como este era possível, e porque estávamos interessados em saber onde colocamos as informações quando programamos". Ainda, que "JANUS nos forçou a prestar atenção para onde nossos bits iam, já que nenhum poderia ser descartado"[43].

$prog$	$::= d^* (\text{procedure } id\ s)^+$	(programa)
d	$::= x \mid x[c]$	(escalar e vetor)
s	$::= x \odot = e \mid x[e] \odot = e \mid x <=> x$	(atribuição, <i>swap</i>)
	$\mid \text{if } e \text{ then } s \text{ else } s \text{ fi } e$	(condicional)
	$\mid \text{from } e \text{ do } s \text{ loop } s \text{ until } e$	(loop)
	$\mid \text{skip}$	(comando vazio)
	$\mid \text{call } id \mid \text{uncall } id$	(chamada de função)
	$\mid s\ s$	(sequência de comandos)
e	$::= c \mid x \mid x[e] \mid e \otimes e$	(expressão)
c	$::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$	(constante inteira)
$\odot$	$::= + \mid - \mid \wedge$	(operador)
$\otimes$	$::= \odot \mid * \mid \&\& \mid <= \mid \dots$	(operador)

Figura 14: Sintaxe simplificada de Janus

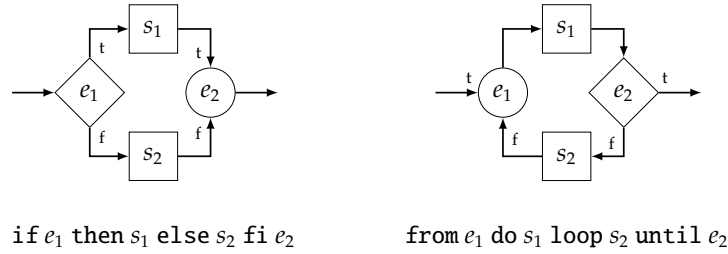


Figura 15: Fluxograma dos controles de fluxo reversíveis de Janus.

No *if*, se a expressão e_1 for verdadeira, a asserção de saída e_2 deve ser verdadeira após a execução de s_1 , e falsa caso contrário. A reversão do *if* executará s_1^{-1} se e_2 for verdadeira, no que e_1 , nesse caso, deverá ser verdadeira. Na reversão do *loop from-do-until*, também acontece a troca de posição entre a condição de entrada e a asserção de saída [9] [26]. A asserção de entrada e_1 deve ser verdadeira na primeira avaliação e falsa nas demais; o teste de saída e_2 deve resultar falso enquanto dura o *loop*. A exigência de que a condição de entrada seja diferente da condição de permanência é o que dá ao *from-do-until* a sua reversibilidade, ao contrário, por exemplo, do *while*, que não é bijetivo.

A Fig. 16 apresenta o código de uma rotina para o cálculo do maior inteiro menor ou igual a $\sqrt{n}$.

6.2.3 Eel: *Energy-efficient language*

O trabalho iniciado por Demaine et al. [12] evoluiu para a formalização da linguagem Eel [64]. A linguagem foi projetada com três níveis de abstração, cada um expondo diferentes graus de complexidade e controle sobre os recursos computacionais. Os diferentes níveis não correspondem a modelos computacionais distintos no sentido

```

procedure main()
  int n
  int root
  n += 25
  call root(n, root)

procedure root(int n, int roo)           //root := floor(sqrt(n))
  local int bit = 1
  from bit = 1 do
    skip
  loop
    call doublebit(bit)
  until (bit * bit) > n
  from (bit * bit) > n do
    uncall doublebit(bit)
    if ((root + bit) * (root + bit)) <= n then
      root += bit
    fi (root / bit)
  loop
    skip
  until bit = 1
  delocal int bit = 1
  n -= root * root

procedure doublebit(int bit)
  local int z = bit
  bit += z
  delocal int z = bit / 2

```

Figura 16: Programa em Janus para cálculo de $\lfloor \sqrt{n} \rfloor$ [43].

clássico, mas a diferentes representações de um mesmo programa ao longo do processo de refinamento, compilação e análise energética. A sintaxe de Eel é reproduzida na Figura 17.

O nível mais baixo conta com operações aritméticas reversíveis, bem como com operações para tratamento de pilha do usuário e a de *log*, pilha especial do sistema e auxiliar da reversão, vista mais à frente. Pilar da reversibilidade, o comando SWAP permite o intercâmbio atômico do valor de um registrador com o de uma posição de memória apontada por outro registrador. Algumas operações irreversíveis são possíveis, com custo energético máximo equivalente à quantidade de bits de uma palavra (w). Operações de saltos incondicionais e condicionais são oferecidas, juntamente com os comandos que devem marcar seus pontos de destino e que indicam de onde estes se originaram: os *comefrom*.

$$\begin{array}{lcl}
 label_1: & \text{GOTOIF}(\boxed{n > 0}, label_2) & \\
 & \dots & \left. \vphantom{\begin{array}{l} label_1: \\ \dots \end{array}} \right\} \text{teste da fase direta} = \text{teste da fase reversa} \\
 label_2: & \text{CMFRMIF}(\boxed{n > 0}, label_1) &
 \end{array}$$

O nível intermediário adiciona definição e chamada a funções, operadores de comparação e os comandos `Log` e `Unroll`, que delimitam blocos de comandos para os quais o compilador gerará automaticamente o código para sua reversão: com o primeiro se iniciam as anotações necessárias; o segundo é a ordem para que seja executada a reversão⁴. Os saltos condicionais já aparecem em duas versões: *protegida* se diz daquele cuja avaliação da expressão de teste se mantém quando da passagem do programa pelo ponto de pouso, que reversamente será um ponto de salto. *Geral* é aquele salto que não dá garantias da manutenção do valor da expressão de teste após sua avaliação primeira. No último caso é necessário que se pague um bit de espaço para que o resultado original do teste seja salvo de modo que a estrutura possa ser revertida⁵. O nível é descrito pelos autores como remanescente do modelo RAM transdicotômico: assume-se, à maneira do modelo, que índices e endereços cabem em uma palavra de tamanho $w = \Theta(\log N)$, onde N denota o tamanho da entrada medido em palavras, o que permite realizar operações aritméticas e acessos à memória em tempo constante.

$ \begin{array}{lcl} label_1: & \text{PGotoIf}(\boxed{n > 0}, \boxed{n > 0}, label_2) & \\ & \dots < n \text{ não é alterada} > \dots & \\ label_2: & \boxed{n > 0} \checkmark & \end{array} $	$ \begin{array}{lcl} label_1: & \text{GotoIf}(\boxed{n > 0}, label_2) & \\ & \dots < n \text{ pode ser alterada} > \dots & \\ label_2: & \boxed{n > 0} ? & \end{array} $
--	--

Com o nível mais alto aparecem os condicionais e os *loops* em suas formas protegidas (`PIf` e `PFor`) e gerais (`If`, `For` e `While`). Da mesma forma, protegida se diz da estrutura na qual as variáveis envolvidas nas expressões de teste não são alteradas pelo corpo da estrutura. Nas formas não protegidas o compilador salva e restaura automaticamente o valor do teste original utilizando a pilha de *log*. O conceito de proteção no *loop* `PFor( init(x), cond(x), incr(x) )` requer que *incr(x)* seja a única modificadora de *x* e que a condição de terminação *cond(x)* não dependa de variável com valor alterado pelo corpo do *loop*. Como deve ser garantido que a quantidade de repetições seja igual nas duas fases, nos *loops* gerais é automaticamente introduzida uma variável auxiliar para controlar o número de iterações, e seu manejo é incluído no código.

⁴A escrita 'à la PISA', espelhada, utilizada nos exemplos de código tem como objetivo que o compilador possua as informações necessárias a geração automática do código reverso.

⁵Lembrar da definição de Copenhagen.

Operação	Descrição
Operações Reversíveis	
ADD(a, b)	$a += b$
SUB(a, b)	$a -= b$
MULT(a, b)	$a *= b$
NEG(a)	$a *= -1$
SWAP(a, b)	$a \leftrightarrow b$
LPUSH(x)	$\log \text{ stack} \leftarrow x$
LPOP(x)	$\log \text{ stack} \rightarrow x$
PUSH(x)	$\text{stack} \leftarrow x$
POP(x)	$\text{stack} \rightarrow x$
Operações Irreversíveis	
MOVE(a, b)	$a = b$
AND(a, b)	$a \wedge = b$
OR(a, b)	$a \vee = b$
Operações de desvio	
GOTO(l)	$pc \leftarrow l$
GOTOIF(e, l)	se $e = \text{true}$, $PC \leftarrow l$
GOTOIFN(e, l)	se $e = \text{false}$, $PC \leftarrow l$
CMFRM(l)	origem em l
CMFRMIF(e, l)	origem em l , se $e = \text{true}$
CMFRMIFN(e, l)	origem em l , se $e = \text{false}$

$prog ::= b$ (bloco)
 $b ::= s^*$ (sequência de comandos)
 $s ::= x \otimes = e \mid x = e$ (atribuição)
 $\quad \mid \text{PGoto}(l)$ (salto protegido)
 $\quad \mid \text{PGotoIf}(e, e, l)$
 $\quad \mid \text{PGotoIfN}(e, e, l)$
 $\quad \mid \text{PLabel}(l)$ (condicional protegido)
 $\quad \mid \text{Goto}(l)$ (salto não protegido)
 $\quad \mid \text{GotoIf}(e, l)$
 $\quad \mid \text{GotoIfN}(e, l)$
 $\quad \mid \text{Label}(l)$
 $\quad \mid \text{Def } q(x, \dots, x) : b$ (definição de função)
 $\quad \mid \text{Call } q(x, \dots, x)$ (chamada de função)
 $\quad \mid \text{Log} : b$ (bloco de log)
 $\quad \mid \text{Unroll } b$ (reversão do log)
 $\quad \mid \text{LogPush}(x)$ (modificação da pilha do log)
 $e ::= c \mid x \mid e \odot e$ (expressão)
 $\otimes ::= + \mid - \mid *$ (operador)
 $\odot ::= \otimes \mid / \mid <= \mid >= \mid \neq \mid ==$ (operador)

$prog ::= b$ (bloco)
 $b ::= s^*$ (sequência de comandos)
 $s ::= x \otimes = e \mid x = e$ (atribuição)
 $\quad \mid \text{PIf}(e) : b \text{ (Else: } b \text{)?}$ (condicional protegido)
 $\quad \mid \text{If}(e) : b \text{ (Else: } b \text{)?}$ (condicional geral)
 $\quad \mid \text{PFor}(s, e, s) : b$ (loop for protegido)
 $\quad \mid \text{For}(s, e, s) : b$ (loop for geral)
 $\quad \mid \text{While}(e) : b$ (loop while geral)
 $\quad \mid \text{Def } q(x, \dots, x) : b$ (definição de função)
 $\quad \mid q(x, \dots, x)$ (chamada de função)
 $\quad \mid \text{Log} : b$ (bloco de log)
 $\quad \mid \text{Unroll} : b$ (reversão do log)
 $e ::= c \mid x \mid e \odot e$ (expressão)
 $\otimes ::= + \mid - \mid *$ (operador)
 $\odot ::= \otimes \mid / \mid <= \mid >= \mid \neq \mid ==$ (operador)

Figura 17: Sintaxe de Eel nos níveis baixo, médio e alto de abstração, sendo: $x \in \text{Variáveis}$, $q \in \text{Funções}$, $c \in \text{Constantes}$.

6.2.4 Outras linguagens reversíveis

A arquitetura BobISA [60] permite 16 registradores de uso geral ($r_0, \dots, r_{15}$) e assume os mesmos 3 de registro geral de PISA: PC, BR e DIR, embora seu conjunto de instruções seja mais simples. Os saltos máximos são de 127 posições de memória, e cada uma contém 16 bits. r_0 também carrega o valor zero, por definição. A implementação física dos saltos foi melhorada com relação à solução dada por Pendulum/PISA.

R foi concebida por Frank e Knight Jr [23] como uma representação intermediária reversível entre linguagens de alto nível e arquiteturas reversíveis como Pendulum. Possui basicamente os mesmos comandos e operadores matemáticos e lógicos de Janus, inclusive o condicional *if* e o *loop* de quantidade de iterações fixas, *for* (equivalente ao *from*). Permite atribuições não destrutivas, *swap* entre variáveis, rotações de bits (bijetivas) e deslocamentos controlados. Vetores são indexados através do operador sublinhado (_), ao invés dos usuais colchetes ([]). A sintaxe de R se baseia em níveis de parênteses como na linguagem LISP.

R00PL estende os princípios de Janus para uma linguagem orientada a objetos [32]. Permite tipos inteiros e referências a objetos definidos por classes, que podem incluir herança e polimorfismo. Além das atribuições reversíveis e trocas, introduz blocos para criação e destruição controlada de objetos. As estruturas de controle seguem o modelo de Janus, com condicionais, laços reversíveis, chamadas e *uncalls* de funções.

6.3 O que temos hoje

Do esquema de possibilidades no intercruzamento das lógicas reversível e irreversível apresentado pela Figura 18, no estado atual da arte, dispomos de:

- Compiladores de Janus, R e R00PL para PISA [2, 23, 32];
- simuladores de PISA, de modo que a reversibilidade possa ser testada através das máquinas existentes: o ambiente de simulação de Vieri foi, na prática, um emulador reversível completo do PISA; e
- emulador de Janus (<https://topps.diku.dk/pirc>);

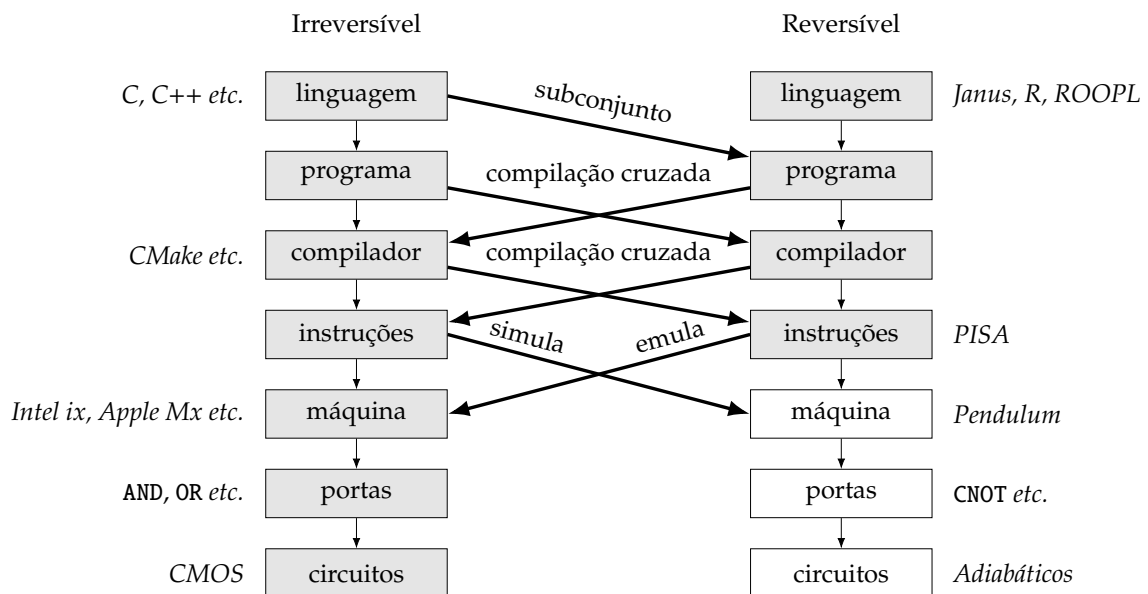


Figura 18: Possibilidades considerando a reversibilidade e a irreversibilidade dos diversos níveis de abstração. Os estágios em branco significam que ainda não foram implementados de forma geral, escalável e comercialmente viável.

6.4 Análise da reversibilidade das principais estruturas de alto nível

O compilador Janus-PISA [2] fornece uma demonstração construtiva da reversibilidade das grandes estruturas de alto nível, a saber: os condicionais, as repetições e chamada de funções. Passemos à sua análise, a partir do código gerado pelo compilador.

Condicionais. (Fig. 19) A verificação inicial de erro encontra seu sentido pelo espelhamento com a parte final do código (erro que ocorre se $r_t \neq 0$). O condicional se inicia com o cálculo da expressão e_1 , de teste para o salto, cujo resultado é armazenado em r_e e salvo em r_t ; assim, o valor intermediário r_e é descomputado⁶. Se $e_1 = \text{false}$, o programa salta para teste_f , com a mecânica oculta já descrita pela Fig. 12, e notando-se que o pouso aponta para a origem do salto. Após a execução do bloco de comandos s_2 , o comando BNE não é acionado, uma vez que $r_t = 0$. Então é verificada a consistência entre a condição de entrada e a condição de saída, ou seja, se o valor da avaliação de e_2 é o mesmo inicialmente calculado para e_1 . De outra forma, se, em teste , $r_t = e_1 = \text{true}$, o bloco s_1 é executado e o programa salta para o teste de asserção. Este compara $r_t = 1$ com a avaliação de e_2 , que também deve ser igual a 1 (true): r_t é zerado. A leitura reversa do código mostra que será executada a inversão do condicional, ou seja:

⁶Essa técnica pode ser classificada entre aquelas que Demaine et al. [12] chamam de *periodic unwinding*, aqui traduzida como 'reverta o mais rápido que puder'. Ao invés de se interpretar o método de Bennett canonicamente, reversões periódicas diminuem em muito o espaço utilizado pelo programa.

if e_2 then s_2 else s_1 fi e_1 .

	BNE r_t, r_0 error	; verificação de erro
	< código para $r_e \leftarrow [e_1]_c$ >	; calcula e_1
	XOR r_t, r_e	; faz $r_t = r_e$
	< código para $r_e \rightarrow [e_1]_c$ >	; descalcula e_1
teste:	BEQ r_t, r_0 teste _f	; salta se $[e_1]_c = 0$
	XORI $r_t, 1$	; zera r_t
	< código para $e_1 = \text{verdadeiro} : s_1$ >	
	XORI $r_t, 1$	; faz $r_t = 1$
asser _v :	BRA asser	; salta
teste _f :	BRA teste	; recebe o salto
	< código para $e_1 = \text{falso} : s_2$ >	
asser:	BNE r_t, r_0 asser _v	; recebe o salto
	< código para $r_e \leftarrow [e_2]_c$ >	; calcula e_2
	XOR r_t, r_e	; zera r_t
	< código para $r_e \rightarrow [e_2]_c$ >	; descalcula e_2
	BNE r_t, r_0 error	; verificação de erro

Figura 19: Tradução do condicional if e_1 then s_1 else s_2 fi e_2 para PISA.

Loop from. (Fig. 20) O comando **from e_1 do s_1 loop s_2 until e_2** , conforme definido em 6.2.2, caracteriza-se por garantir que a avaliação final de e_2 determina univocamente o número de iterações executadas, permitindo a reconstrução do estado inicial na execução reversa.

A tradução para PISA deve preservar três propriedades: determinismo para frente, determinismo para trás e restauração do estado auxiliar ao término do bloco. O registrador r_t atua como bit de fase, armazenando temporariamente o resultado da avaliação da expressão de teste e permitindo que a simetria entre as fases direta e reversa seja mantida sem perda de informação.

A reversibilidade decorre do fato de que cada avaliação de expressão é imediatamente descomputada, que o valor de r_t codifica univocamente o estado do teste e que o salto condicional é estruturado de modo que o ponto de pouso contenha a informação necessária para a reconstrução do fluxo de controle.

Na execução reversa, a condição e_2 passa a exercer o papel de teste de entrada, enquanto e_1 torna-se condição de parada, reproduzindo exatamente a forma sintática:

from e_2 do s_2^{-1} loop s_1^{-1} until e_1 .

Chamadas de funções. (Fig. 21) O salto incondicional BRA pousa na instrução SWAPBR, que guarda o deslocamento do salto em r_{ro} . O retorno será pelo mesmo ponto, quando a execução da mesma instrução devolverá o deslocamento ao registrador BR, promovendo o salto incondicional para a origem da chamada. Mas isso ocorre não sem a inversão do sinal do deslocamento pela instrução seguinte e seu empilhamento. O bloco correspondente à função em si é executado. Como estratégia para que a execução

```

        XORI rt 1                ; faz rt = 1
entrada: BEQ rt r0 asser          ; recebe o salto
        < código para re ← [e1]c > ; calcula e1
        XOR rt re                ; faz rt = re
        < código para re → [e1]c > ; descalcula e1
        BNE rt r0 error          ; verificação de erro
        < código de s1 >
        BNE rt r0 error          ; verificação de erro
        < código para re ← [e2]c > ; calcula e2
        XOR rt re                ; rt = [e2]c
        < código para re → [e2]c > ; descalcula e2
teste:  BNE rt r0 saída          ; sai se [e2]c = 1
        < código de s2 >
asser:  BRA entrada              ; salta para o topo
saída:  BRA teste                ; recebe o salto de saída
        XORI rt 1                ; zera rt

```

Figura 20: Tradução do *loop* from e_1 do s_1 *loop* s_2 until e_2 para PISA.

encontre o ponto de retorno, um salto incondicional ‘para trás’ encontra os comandos que desempenham o deslocamento a ser utilizado.

Como a reversão altera o sentido do passo do PC ($\text{dir} = -1$), a deschamada (*uncall*) inverte o sentido das flechas da figura e a interpretação das instruções. A simetria do código faz com que o *framework* de *call* e *uncall* executem as mesmas instruções, com a exceção da substituição de f por f^{-1} .

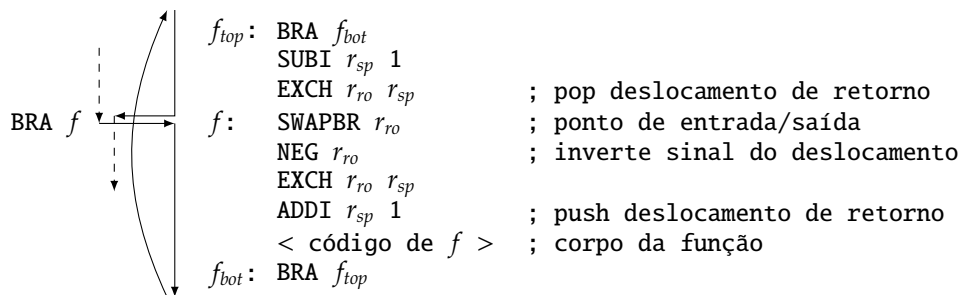


Figura 21: Tradução de chamada de função em PISA.

6.5 Adicionando reversibilidade à linguagem de pseudocódigo

A fim de alcançarmos o objetivo do próximo capítulo — apresentar versões semirreversíveis para algoritmos clássicos — tecemos alguns comentários sobre a adição da reversibilidade à linguagem de pseudocódigo e outros pontos fundamentais.

1. Não ao paradigma pendular. A computação reversível vem sendo desenvolvida desde a década de 1960, mas arquiteturas reversíveis de propósito geral ainda não se

consolidaram como tecnologia prática. Pendulum e seu conjunto de instruções, PISA, embora constituam a base teórica sobre a qual se apoia grande parte da produção no campo, permanecem no âmbito da simulação.

O que se observa, por exemplo, com o compilador Janus-PISA, é a incorporação da ideia de uma máquina cuja reversibilidade decorre da execução inversa integral do próprio programa, instrução a instrução, a partir do momento em que o registrador *dir* altera o sentido do passo do *program counter*: a execução passa, então, a percorrer o código em sentido inverso.

Entretanto, à luz das limitações práticas discutidas no Capítulo 4, adotaremos a mesma perspectiva de Demaine et al. [12]:

Nosso computador [semirreversível teórico] possui somente um simples modo de operação, sempre incrementando o *program counter* com cada instrução. A reversão das operações vêm com a escrita da operação inversa em uma seção posterior de código, ao invés de termos um modo reverso separado que percorre o programa em sentido inverso, invertendo tais operações.

2. Da simetria estrutural à garantia semântica da reversibilidade. Em uma arquitetura na qual o contador de programa e o relógio avançam monotonicamente, a reversão da computação não ocorre por inversão temporal do fluxo de execução, como no modelo pendular, mas por meio da definição explícita de uma fase inversa. Tal arquitetura relaxa a exigência de simetria estrutural entre os trechos direto e reverso do programa, bastando que a composição das duas fases constitua uma transformação bijetiva sobre o conjunto de estados alcançáveis da máquina. Esse relaxamento amplia a liberdade na escrita do código reverso, permitindo reorganizações estruturais, reavaliações condicionais e trajetórias distintas no espaço de estados, desde que nenhuma informação seja destruída e que o estado final determine univocamente o estado inicial. Desloca-se, assim, o critério de reversibilidade para o nível lógico-funcional: a preservação da informação deve ser garantida semanticamente, e não apenas pela simetria operacional do fluxo de controle.

3. Não utilização de pilha de log. Com finalidade didática, optamos por não ocultar a reversão do código por meio das construções *Log* e *Unroll*. Todo algoritmo será apresentado com a fase direta explicitada, seguida de sua fase reversa igualmente escrita de forma explícita.

4. Mera indicação das condições de saída. Em consonância com o item anterior e com o deslocamento semântico discutido no item 2, as asserções de saída nos con-

dicionais reversíveis serão indicadas como comentários entre chaves. Quando houver simetria estrutural entre as fases, tal condição reaparecerá, com mesmo domínio ou expressão equivalente, como teste do condicional correspondente à sua reversão.

5. Modelo computacional e tipo. Como em Eel, assume-se que a palavra possui tamanho suficiente para que as operações e acessos à memória executem em tempo $O(1)$. As variáveis simples, vetores multidimensionais e pilha utilizam o tipo básico inteiro, representável em w bits, no intervalo $[-2^{w-1}, \dots, 2^{w-1} - 1]$.

6. Controle de fluxo. O salvamento de informação necessária para a reversibilidade de condicionais não protegidos fica a cargo do programador. A estrutura de repetição é tomada emprestada de Janus, com sintaxe `from  $e_{in}$  do  $s$  until  $e_{out}$` . A condição de entrada é o valor de verdade da expressão e_{in} , quando o bloco de comando s é executado até que se observe a condição expressa por e_{out} . A expressão e_{in} é avaliada apenas antes da primeira iteração. O incremento ou decremento da variável em jogo no laço deve ser providenciado pelo programador⁷, bem como a gerência das informações em caso de não proteção. Visando à simplificação, as variáveis são consideradas como globais; a passagem de argumentos para as funções se dá por valor, de modo que a função deve preservar a reversibilidade global do estado compartilhado.

7. Atribuição e movimentação de memória. As atribuições não destrutivas utilizam operadores binários que alteram reversivelmente o valor de uma variável x a partir de uma expressão e : `$x \leftarrow x \odot e$` (ou: `$x \odot \leftarrow e$`), na qual e não pode depender de x . Axelsen [2] mostra a reversibilidade das atribuições através da compilação de Janus para PISA. As movimentações para posições de memória zeradas não dissipam e então as atribuições `$x \leftarrow 5$` e `$x^+ \leftarrow 5$` são equivalentes desde que esteja garantido que $x = 0$ no momento da atribuição. Adotamos a convenção de sublinhar a variável com valor igual a zero quando esta recebe uma atribuição, como em `$x$   $\leftarrow 5$` [12]. A permuta não destrutiva de valores entre duas variáveis x e y escreve-se `$x \leftrightarrow y$` . A semântica das funções push e pop é a da permuta de valores entre a variável e a pilha.

Quaisquer atribuições fora dos padrões descritos incorre na dissipação correspondente à perda de até w bits de informação. O fato também vale para a utilização de operações lógicas destrutivas.

⁷Note que, aqui, o `from-do-until`, ao contrário do seu similar em Janus, possui somente um bloco de comandos.

7 Versões reversíveis para algoritmos clássicos

Seguindo a linha de raciocínio exposta nos capítulos anteriores e utilizando a linguagem de pseudocódigo acrescida de recursos de reversibilidade (ver seção 6.5), apresentamos versões logicamente reversíveis de implementações clássicas originalmente descritas no modelo irreversível.

Através dessas soluções esperamos que fique um pouco mais clara a mecânica da combinação de primitivas reversíveis, de modo que o pensamento se acostume a essa nova forma de proceder que, ao mesmo tempo em que detecta proibições, se prontifica mais facilmente a contorná-las.

Nas seções seguintes lançamos uma lupa sobre os problemas, sua definição, as estruturas de dados utilizadas, a estratégia de reversibilização adotada, e tecemos comentários complementares sobre o que porventura não tenha sido coberto anteriormente.

Todas as soluções encontradas são logicamente reversíveis, implicando custo energético nulo no modelo de Landauer adotado.

Os algoritmos estudados foram:

- Subsequência Comum Mais Longa (Longest Common Subsequence – LCS);
- Floyd-Warshall; e
- Radix Sort.

7.1 Subsequência Comum Mais Longa (Longest Common Subsequence – LCS)

Dada uma sequência $S = \langle s_1, s_2, \dots, s_n \rangle$, uma subsequência de S é obtida a partir de S com a remoção de zero ou mais elementos. Ou seja, é uma sequência com um subconjunto de elementos, respeitando a ordem original dos elementos. Assim, se $S = ABCDEF$ então $ABDE$ é uma subsequência de S , mas $AEBD$ não é. A subsequência comum a S_1 e S_2 mais longa LCS é aquela que é subsequência tanto de S_1 quanto de S_2 , e que tem o maior comprimento possível [10].

O problema de determinar a LCS encontra aplicação na biologia molecular e mais especificamente na avaliação do grau de similaridade entre DNA, que nada mais são abstratamente que sequências de bases nitrogenadas representadas por letras.

7.1.1 Regras de Recorrência

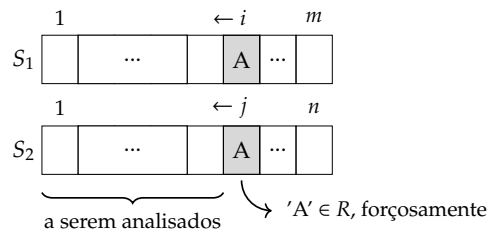
Dadas duas *strings* S_1 e S_2 , de tamanhos respectivos m e n , definimos $LCS[i, j]$ como a quantidade de caracteres da subsequência comum mais longa considerando os i primeiros caracteres de S_1 e os j primeiros caracteres de S_2 .

As regras de recorrência são:

$$LCS[i, j] = \begin{cases} LCS[i-1, j-1] + 1, & \text{se } i, j \neq 0 \text{ e } S_1[i] = S_2[j] \quad (1) \\ \max \{LCS[i-1, j], LCS[i, j-1]\}, & \text{se } i, j \neq 0 \text{ e } S_1[i] \neq S_2[j] \quad (2) \\ 0, & \text{se } i = 0 \text{ ou } j = 0 \quad (3) \end{cases}$$

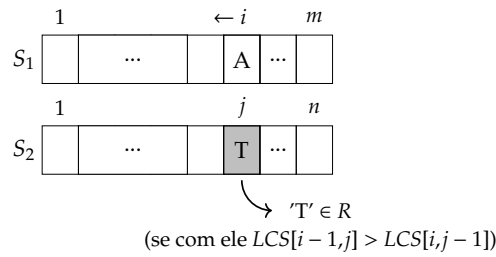
Note que as regras de recorrência visam primariamente obter a contagem estrutural dos caminhos e, assim, estabelecer o tamanho da subsequência mais longa. A obtenção do resultado ótimo, a própria subsequência, é um subproduto secundário a ser computado em seguida.

A análise das regras informa como é obtida a própria subsequência. $S_1[i] = S_2[j]$ indica que foi encontrado um caractere que deve compô-la forçosamente; neste momento, $S_1[i]$ (ou $S_2[j]$) é concatenado à *string* de resultado R , e sabemos que o tamanho da LCS até este ponto ($[i, j]$) é o tamanho da LCS considerando os caracteres $S_1[1] \dots S_1[i-1]$ e $S_2[1] \dots S_2[j-1]$, somado de 1.

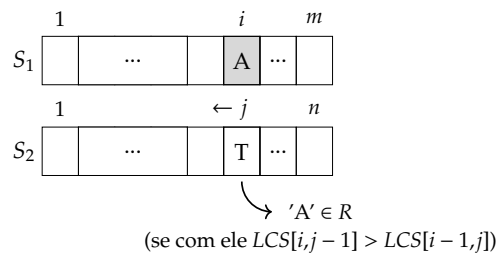


Por outro lado, se $S_1[i] \neq S_2[j]$, então o tamanho da LCS até este ponto é o maior resultado entre:

- $LCS[i-1, j]$, ou seja, considerando os caracteres $S_1[1] \dots S_1[i-1]$ e $S_2[1] \dots S_2[j]$: pois assume-se que $S_1[i] \notin R$ e então o elemento é descartado e $S_2[j]$ pode ainda eventualmente ser concatenado ao resultado; ou



- $LCS[i, j-1]$, ou seja, considerando os caracteres $S_1[1] \dots S_1[i]$ e $S_2[1] \dots S_2[j-1]$: pois assume-se que $S_2[j] \notin R$ e então o elemento é descartado e $S_1[i]$ pode ainda eventualmente ser concatenado ao resultado.



7.1.2 Solução recursiva

Das regras de recursão extrai-se automaticamente uma solução recursiva para determinar o tamanho da LCS, como a do Algoritmo 1.

7.1.3 Programação Dinâmica

A abordagem do problema pela força bruta leva a uma complexidade de tempo inaceitável, o que faz com que o problema LCS seja elegível para a resolução através da

Algoritmo 1: LCS: solução recursiva

```

1  $LCS(i, j)$ 
2 se  $i = 0$  ou  $j = 0$  então retorna 0
3 se  $S_1[i] = S_2[j]$  então retorna  $LCS(i - 1, j - 1) + 1$ 
4 se  $LCS(i, j - 1) > LCS(i - 1, j)$  então retorna  $LCS(i, j - 1)$ 
5 retorna  $LCS(i - 1, j)$ 
6 fim-LCS

```

programação dinâmica (PD). Isso porque, se a execução for executada da forma tradicional de cima para baixo (*top-down*), a partir da chamada $LCS(m, n)$, esta põe em marcha a revisitação e recálculo de subproblemas que faz com que o tempo cresça assintoticamente de forma exponencial, como exemplificado pela Figura 22.

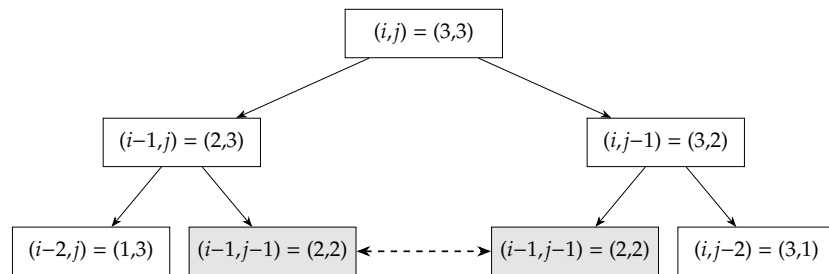


Figura 22: Explosão da quantidade de cálculos na solução recursiva: os recálculos de subproblemas são destacados em fundo cinza.

O cálculo da complexidade utiliza a regra 2, do pior caso, no qual não há coincidência de caracteres, e conta a quantidade de nós da árvore de recursão a partir da raiz $LCS[m, n]$. Tanto o ramo de $LCS[i, j - 1]$ quanto o de $LCS[i - 1, j]$ são executados para determinar o maior valor entre eles, sendo constante o custo associado a cada nó. Como a cada passo os índices i e j são decrescidos de uma unidade, a altura da árvore é $m + n$. Dado essa árvore é binária completa de altura $m + n$, que possui $2^{m+n+1} - 1$ nós, obtemos a complexidade de tempo de $O(2^{m+n})$ ¹.

A resolução de baixo para cima (*bottom-up*) proporcionada pela PD, por um lado, lança mão da memorização (*memoization*) do cálculo do tamanho da LCS para cada combinação possível dos valores de i e de j , o que evita os recálculos. Por outro, reduz o algoritmo ao preenchimento da matriz bidimensional $m \times n$, observadas as regras de recorrência que informam: as posições para as quais o valor será igual a zero, mas também a dependência que o cálculo do valor de uma posição genérica de $LCS[i, j]$ carrega em relação ao prévio cálculo das posições $LCS[i - 1, j - 1]$, $LCS[i - 1, j]$

¹Para sermos mais precisos, o número de folhas dessa árvore corresponde ao número de sequências possíveis de reduções de i e j , que é dado por $\binom{m+n}{m}$. Como $\binom{m+n}{m} \in \Theta(2^{m+n} / \sqrt{m+n})$, a complexidade resultante é $\Theta(2^{m+n})$.

e $LCS[i, j - 1]$ (ver Figura 23). Tal dependência faz com que o preenchimento da matriz possa ser executado indiferentemente de cima para baixo e da esquerda para a direita, ou da esquerda para a direita e de cima para baixo. A quantidade de caracteres da LCS será obtida no cálculo de $LCS[m, n]$. A partir dessa posição, a aplicação da regra de recorrência em sentido reverso faz com que os caracteres da subsequência, do último para o primeiro, sejam revelados para compor a *string* de resultado, R . É o que mostra o Algoritmo 2, que tem complexidades de tempo e de espaço iguais a $\Theta(m.n)$.

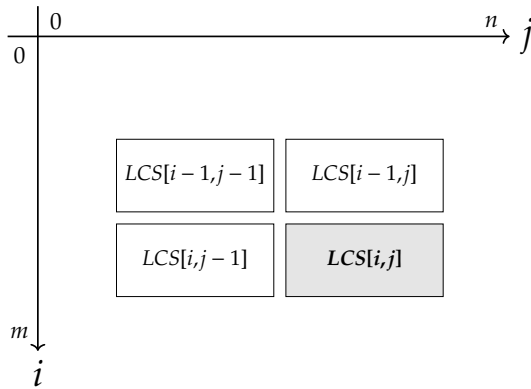


Figura 23: Matriz da programação dinâmica destacando a dependência de $LCS[i, j]$ em relação aos outros valores, que devem estar calculados previamente.

7.1.4 Dificuldades encontradas na reversibilização

O cálculo da LCS através de PD é bijetivo no que se refere à obtenção de $LCS[m, n]$, não necessitando de estratégias adicionais para sua reversibilização. Não há destruição de informação, pois que o coração do problema é o preenchimento da matriz e este é levado a cabo através de atribuições a posições de memória zeradas, procedimento que, como vimos, não incorre em custo energético. Se $\underline{LCS[i, j]} \leftarrow x$ equivale a $LCS[i, j]^+ \leftarrow x$, então a reversão ocorrerá com a aplicação da subtração, operação reversa.

No entanto, a extração da *substring* propriamente dita conforme o Algoritmo 2 enfrenta a questão da reversão final dos índices i e j , uma vez que não necessariamente estes chegam igualmente a zero ao final do procedimento. Para que tal se verifique, o procedimento de extração deve continuar quando um ou outro índice for igual a zero, tendo como condição de parada que tanto i quanto j sejam iguais a zero. Desta forma, ambos os valores dos índices sendo conhecidos, é possível reconfigurá-los iniciar corretamente a descomputação da matriz da PD. Ao invés de utilizarmos uma *string* de resultado, preferimos disponibilizar os caracteres que compõem a LCD na pilha do sistema, de forma que o topo desta conterá o primeiro caractere e assim por diante. A

variável r conterá o tamanho da LCS.

Algoritmo 2: LCS com programação dinâmica

```

1   $LCS(S_1, S_2, m, n, R)$ 
2  para todo  $i \in [0, m]$ 
3     $LCS[i, 0] \leftarrow 0$ 
4  fim-para
5  para todo  $j \in [0, n]$ 
6     $LCS[0, j] \leftarrow 0$ 
7  fim-para
8  para todo  $i \in [1, m]$ 
9    para  $j \in [1, n]$ 
10     se  $S_1[i] = S_2[j]$  então
11        $LCS[i, j] \leftarrow LCS[i - 1, j - 1] + 1$ 
12     senão
13       se  $LCS[i - 1, j] > LCS[i, j - 1]$  então
14          $LCS[i, j] \leftarrow LCS[i - 1, j]$ 
15       senão
16          $LCS[i, j] \leftarrow LCS[i, j - 1]$ 
17     fim-se
18   fim-se
19 fim-para
20 fim-para
21  $\{i = m, j = n\}$ 
22 enquanto  $i > 0$  ou  $j > 0$ 
23   se  $S_1[i] = S_2[j]$  então
24     insere  $S_1[i]$  no início de  $R$ 
25      $i \leftarrow i - 1, j \leftarrow j - 1$ 
26   senão
27     se  $LCS[i - 1, j] > LCS[i, j - 1]$  então
28        $i \leftarrow i - 1$ 
29     senão
30        $j \leftarrow j - 1$ 
31   fim-se
32 fim-se
33 fim-enquanto
34 retorna  $R$ 
35 fim-LCS

```

Os condicionais são do tipo *protegido*, ou seja, as posições de memória envolvidas nos testes não são alteradas durante a execução dos blocos de código dos condicionais. Os testes envolvem posições das *strings* S_1 e S_2 , enquanto que os valores alterados nos comandos referem-se a posições da matriz. A reversão dos *loops* é obtida a partir da inversão das condições de entrada e saída em relação aos da fase direta e a inversão das operações internas, assegurando-se que eles executem a mesma quantidade de vezes nas duas direções.

Na inicialização da matriz não é necessário o preenchimento das posições que devem ser zeradas em função da regra quando $i, j = 0$, uma vez que na computação

reversível a máquina deve disponibilizar a memória inicialmente zerada por conta de seus princípios.

O que descrevemos acima encontra-se explicitado no Algoritmo 3.

O tamanho da entrada é $\Theta(m + n)$, de modo que o tamanho da palavra, no modelo transdicotômico, é $\Theta(\log(m + n))$. Dessa forma, consegue-se endereçar qualquer ponto da matriz da PD em tempo constante, $O(1)$.

A complexidade de tempo, como esperado, é a mesma da versão original implementada com a PD: $\Theta(m \cdot n)$. A complexidade de espaço também é $\Theta(m \cdot n)$, em função da matriz ser bidimensional. Como o algoritmo é reversível, a complexidade de energia é 0, considerando que só será pago o preço energético das variáveis de entrada e de saída – gastos que são desconsiderados pela computação reversível.

O fato de a versão reversível não apresentar aumento na complexidade de espaço sugere que, em certos esquemas de PD nos quais cada entrada da matriz é preenchida uma única vez e jamais sobrescrita, o mapeamento global composto pelas duas *strings* e a matriz, pode preservar informação suficiente para caracterizar uma transformação bijetiva. Nesses casos, a reversibilização pode ser obtida sem a necessidade de armazenamento auxiliar adicional.

Algoritmo 3: LCS reversível

```

1   $LCS$ 
2  {  $LCS[m, n], i, j = 0$  }
3   $\underline{i} \leftarrow 1, \underline{j} \leftarrow 1$ 
4  from  $i = 1$  do
5      from  $j = 1$  do
6          if  $S_1[i] = S_2[j]$  then  $\underline{LCS}[i, j] \leftarrow LCS[i - 1, j - 1] + 1$ 
7          else
8              if  $LCS[i - 1, j] > LCS[i, j - 1]$  then
9                   $\underline{LCS}[i, j] \leftarrow LCS[i - 1, j]$ 
10             else
11                  $\underline{LCS}[i, j] \leftarrow LCS[i, j - 1]$ 
12             end-if {  $LCS[i - 1, j] > LCS[i, j - 1]$  }
13         end-if {  $S_1[i] = S_2[j]$  }
14          $j^+ \leftarrow 1$ 
15     until  $j = n + 1$ 
16      $j^- \leftarrow n, i^+ \leftarrow 1$ 
17 until  $i = m + 1$ 
18 .....
19  $\underline{r} \leftarrow LCS[m, n]$ 
20  $i^- \leftarrow 1, j^+ \leftarrow (n - 1)$ 
21 from  $i = m$  and  $j = n$  then
22     if  $i = 0$  then  $j^- \leftarrow 1$ 
23     else
24         if  $j = 0$  then  $i^- \leftarrow 1$ 
25         else
26             if  $S_1[i - 1] = S_2[j - 1]$  then  $\text{push}(S_1[i - 1])$ ,  $i^- \leftarrow 1, j^- \leftarrow 1$ 
27             else
28                 if  $LCS[i - 1, j] > LCS[i, j - 1]$  then  $i^- \leftarrow 1$ 
29                 else  $j^- \leftarrow 1$  end-if
30             end-if
31         end-if
32     end-if
33 until  $i = 0$  and  $j = 0$ 
34  $i^+ \leftarrow m + 1, j^+ \leftarrow 1$ 
35 .....
36 from  $i = m + 1$  do
37      $i^- \leftarrow 1, j^+ \leftarrow n$ 
38     from  $j = n + 1$  do
39         if  $S_1[i] = S_2[j]$  then  $LCS[i, j]^- \leftarrow LCS[i - 1, j - 1] + 1$ 
40         else
41             if  $LCS[i - 1, j] > LCS[i, j - 1]$  then
42                  $LCS[i, j]^- \leftarrow LCS[i - 1, j]$ 
43             else
44                  $LCS[i, j]^- \leftarrow LCS[i, j - 1]$ 
45             end-if {  $LCS[i - 1, j] > LCS[i, j - 1]$  }
46         end-if {  $S_1[i] = S_2[j]$  }
47     until  $j = 1$ 
48 until  $i = 1$ 
49  $j^- \leftarrow 1, i^- \leftarrow 1$ 
50 end-LCS

```

7.2 Floyd-Warshall

O algoritmo de Floyd-Warshall [18] é uma solução para encontrar os caminhos mais curtos (ou menos custosos) em um grafo direcionado com pesos positivos ou negativos nas arestas, vedados os ciclos negativos².

Seja G um grafo direcionado com pesos nas arestas, onde $V(G) = \{1, \dots, n\}$. A ideia é a de que a menor distância entre dois vértices u e v , que de início é dado pelo peso da aresta (u, v) , possa ser atualizada com a distância do caminho mais curto entre u e v dentre os que incluem um vértice intermediário k e utilizam somente vértices entre 1 e k como vértices internos. Ou seja, se a menor distância entre u e k somada com a menor distância entre k e v for menor do que a menor distância entre u e v permitindo apenas vértices intermediários em $\{1, 2, \dots, k-1\}$, a soma atualiza o valor inicial. Note que: (a) o menor custo entre um par de vértices deve ser buscado testando-se todos os vértices como o possível intermediário de maior índice; e (b) a definição é recursiva, significando que tanto o menor custo entre u e k quanto o entre k e v recaem em subproblemas. A mecânica básica do cálculo do menor custo é apresentada na Figura 24.

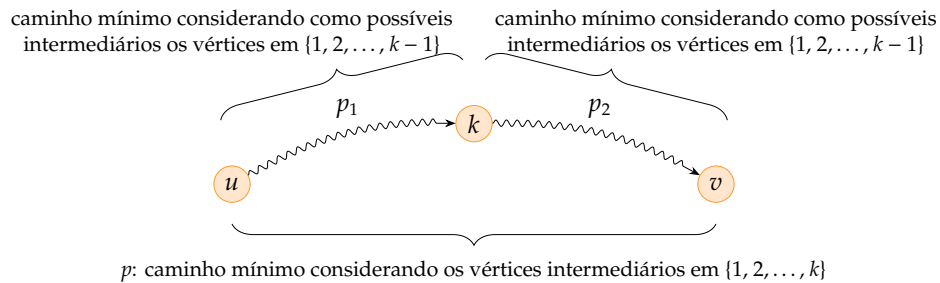


Figura 24: Caso básico do algoritmo de mínimo custo. p_1 é a menor distância possível entre u e k , considerando todos os outros $k-1$ possíveis intermediários; p_2 significa o mesmo entre k e v . p é o menor caminho entre u e v , considerando todos os k possíveis vértices intermediários.

O algoritmo de Floyd-Warshall se desenvolve a partir de uma matriz D^0 dos pesos das arestas de um grafo G de n vértices, onde D^0 é uma matriz $n \times n$ tal que cada posição $D^0[u, v]$ contém:

- 0, se $u = v$;
- o peso da aresta (u, v) , se $(u, v) \in V(G)$;
- ∞ , se $(u, v) \notin G$.

²Um ciclo negativo é definido como um ciclo no qual a soma dos pesos de todas as arestas que o compõem resulta em um valor menor que zero.

Note que ∞ na verdade representa um valor suficientemente grande. No contexto de caminhos mínimos, podemos considerar ∞ como o somatório dos pesos positivos das arestas acrescido de um.

Dado D^0 , definimos D^k , para $k \in \{1, \dots, n\}$, como uma matriz $n \times n$ onde para $u, v \in \{1, \dots, n\}$ temos que a célula $D^k[u, v]$ armazena o custo do caminho mais barato entre u e v que utiliza somente vértices entre $\{1, \dots, k\}$ como intermediários. Logo, D^n é a matriz contendo a informação dos custos dos caminhos mínimos entre todos os pares de vértices de G .

A regra de recorrência para computação dos caminhos mínimos é dada por:

$$D^k[u, v] = \min \{D^{k-1}[u, v], D^{k-1}[u, k] + D^{k-1}[k, v]\}, \text{ onde } k \in \{1, \dots, n\}$$

7.2.1 Solução recursiva

A partir da recorrência clássica pode-se extrair um algoritmo recursivo como o Algoritmo 4, no qual n é o número de vértices.

Algoritmo 4: Floyd-Warshall: solução recursiva

```

1  FW(k, u, v)
2      retorna min {FW[k - 1, u, v], FW[k - 1, u, k] + FW[k - 1, k, v]}
3  fim-FW

4  para todo u ∈ [1, n]
5      para todo v ∈ [1, n]
6          D[u, v] ← FW(n, u, v)
7      fim-para
8  fim-para

```

A estrutura utilizada é uma matriz bidimensional cujas células capturam o valor final da distância entre cada par de vértices ($D[u, v]$). A árvore de recursão para cálculo desses valores faz surgir a 3ª dimensão, a de k , como sua profundidade n , que equivale ao espaço gasto no cálculo dos valores intermediários. A complexidade de espaço total dessa versão é $\Theta(n^2)$.

A utilização da abordagem *top-down* leva à explosão do tempo de execução pelo recálculo de subproblemas (Figura 25). O passo base de cada ramo é o acesso aos valores da matriz de distâncias D^0 . Para cada par de vértices é executada uma árvore que, segundo a regra de recursão se escreve:

$$T(k) = T(k-1) + 2T(k-1) + O(1) = 3T(k-1) + O(1)$$

Leia-se: a cada nível k cujo custo é sempre constante, toma-se uma decisão que realiza 3 cálculos recursivos. Por substituição progressiva, o custo parcial de tempo é:

$$T(k) = 3(3T(k-2) + O(1)) + O(1) = 3^2T(k-2) + 3^1O(1) + 3^0O(1)$$

...

$$T(k) = 3^kT(0) + \sum_{i=0}^{k-1} 3^iO(1) = 3^k + \frac{3^k - 1}{2} = \Theta(3^k)$$

Como a árvore é executada para todos os pares, ou seja, n^2 vezes, então a complexidade de tempo desta solução recursiva é $\Theta(n^2 \cdot 3^n)$.

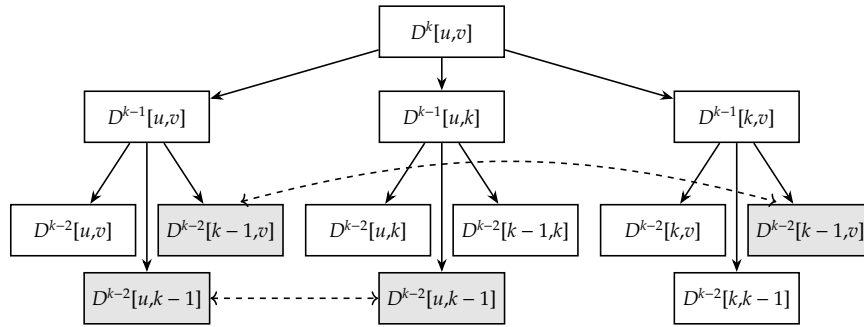


Figura 25: Explosão do tempo de execução pelo recálculo de subproblemas em Floyd-Warshall na abordagem *top-down*: as repetições estão destacados em fundo cinza.

7.2.2 Programação dinâmica

A resolução do algoritmo por programação dinâmica (PD) elimina o recálculo de subproblemas, por iniciar a computação pelas instâncias sem dependência e fornecê-las gradativamente às situadas nos andares superiores da recursão. O problema torna-se tratável pela redução de sua complexidade de tempo ao nível polinomial: $\Theta(n^3)$. A dimensão referente a k surge como o número da iteração na qual a matriz bidimensional de distâncias D está sendo inteiramente percorrida para atualização dos seus valores: com isso, cada posição $D[u,v]$ é sobrescrita n vezes. Para facilitar essa visualização, escrevemos D^k para referenciarmos a k -ésima iteração de atualização das posições de D . A complexidade de espaço se mantém como $\Theta(n^2)$.

Não há ordem preferencial para o preenchimento ou atualização de uma matriz D^k pois, segundo a regra de recorrência, $D^k[u,v]$ depende de $D^{k-1}[u,v]$, $D^{k-1}[u,k]$ e

$D^{k-1}[k,v]$, e todas as células da matriz D^{k-1} se encontram disponíveis no início da iteração correspondente a D^k (Figura 26). Adotamos a orientação da esquerda para a direita e de cima para baixo (por linha). Cada célula é recalculada n vezes. A resposta ao problema será a matriz D^n .

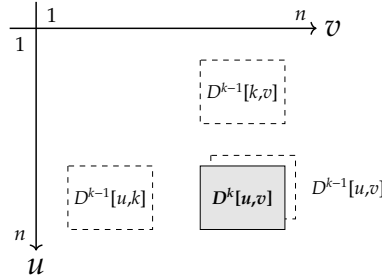


Figura 26: Matriz da programação dinâmica destacando a dependência de $D^k[u,v]$ em relação aos outros valores, que já estão disponíveis em função do cálculo prévio de D^{k-1} .

A solução não precisa utilizar uma matriz tridimensional porque que o laço mais externo do algoritmo é o da variação de k . Dessa forma, não há risco que durante uma iteração k , a atualização de uma célula consulte outras células que possam ter sido já sobrescritas nessa mesma iteração, dado que, pela regra de recorrência:

$$\begin{aligned} D^k[u,k] &\leftarrow \min \{ D^{k-1}[u,k], D^{k-1}[u,k] + \overset{0}{D^{k-1}[k,k]} \} \Rightarrow D^k[u,k] \leftarrow D^{k-1}[u,k] \\ D^k[k,v] &\leftarrow \min \{ D^{k-1}[k,v], \overset{0}{D^{k-1}[k,k]} + D^{k-1}[k,v] \} \Rightarrow D^k[k,v] \leftarrow D^{k-1}[k,v] \end{aligned}$$

Ou seja, mesmo que os valores das células consultadas já tenham sido recalculados na mesma iteração, estes não foram alterados em relação aos da iteração anterior³.

O Algoritmo 5 descreve a versão clássica da programação dinâmica de Floyd-Warshall:

Algoritmo 5: Floyd-Warshall

```

1  FW( $D^0, n$ )
2  para todo  $k \in [1, n]$  faça
3      para  $u \in [1, n]$  faça
4          para  $v \in [1, n]$  faça
5               $D^k[u, v] \leftarrow \min \{ D^{k-1}[u, v], D^{k-1}[u, k] + D^{k-1}[k, v] \}$ 
6          fim-para
7      fim-para
8  fim-para
9  fim-FW
```

Sua complexidade de espaço é de $\Theta(n^2)$, e a de tempo, $\Theta(n^3)$.

³Tal raciocínio não valeria se pudesse ser encontrado um caminho com valor negativo que ligasse um vértice a ele mesmo, mas os ciclos negativos são vedados.

A seguir discutiremos as dificuldades enfrentadas na produção de uma versão reversível do Floyd-Warshall que visa manter a complexidade de espaço próxima da versão não reversível original.

7.2.3 Dificuldades encontradas na reversibilização

Para produzir uma versão reversível do Algoritmo 5 é necessário que:

- não haja sobrescrita que apague informações; e
- todas as operações realizadas sejam bijetivas, individualmente e em bloco.

O Floyd-Warshall se baseia na atualização da distância entre cada par de vértices, o que sugere uma solução natural para sua computação reversível: a de apenas transformar cada iteração k em mais uma dimensão da matriz D da PD, que se torna, assim, tridimensional. Assim, D^k se torna dependente dos valores localizados em D^{k-1} e cada célula encontra sua posição previamente zerada⁴.

Logo, após a computação dos caminhos mínimos, a matriz D^k pode ser zerada utilizando os valores de D^{k-1} , até que D^1 seja zerada a partir da matriz de pesos D^0 .

A solução reversível que segue a estratégia anterior é apresentada pelo Algoritmo 6. A fase direta consiste basicamente no encadeamento de três *loops from-do-until*. A fase reversa decorre da aplicação dos comandos inversos daqueles da fase direta.

O tamanho da entrada é $O(n^2)$, o algoritmo executa em tempo $\Theta(n^3)$ com gasto de espaço $\Theta(n^3)$ e energia zero, os mesmos valores citados por Demaine *et al.* [12] e por Frank and Knight Jr [23].

7.2.4 Revisitando a questão do espaço

No algoritmo de Floyd-Warshall cada entrada $D^k[u,v]$ depende apenas de três valores da camada anterior, mas, globalmente, cada matriz intermediária D^k depende funcionalmente da matriz D^{k-1} inteira. O grafo de dependências de subproblemas forma, assim, uma sequência de camadas de largura $\Theta(n^2)$, cujo corte mínimo entre níveis sucessivos também tem tamanho $\Theta(n^2)$.

Torna-se, assim, aparentemente inevitável o armazenamento de $\Theta(n^3)$ informações ao longo da execução para garantir a reversibilidade, pois não há subconjunto inter-

⁴Preferimos manter a notação para que o aspecto temporal da dimensão não se perca.

Algoritmo 6: *FloydWarshall* reversível

```

1  FloydWarshall
2  {  $k, u, v, D^n[n,n] = 0$  }
3   $\underline{k} \leftarrow 1, \underline{u} \leftarrow 1, \underline{v} \leftarrow 1$ 
4  from  $k = 1$  do
5      from  $u = 1$  do
6          from  $v = 1$  do
7              if  $D^{k-1}[u,v] > (D^{k-1}[u,k] + D^{k-1}[k,v])$  then
8                   $\underline{D^k[u,v]} \leftarrow D^{k-1}[u,k] + D^{k-1}[k,v]$ 
9              else
10                  $\underline{D^k[u,v]} \leftarrow D^{k-1}[u,v]$ 
11             end-if
12              $v^+ \leftarrow 1$ 
13             until  $v = n + 1$ 
14              $u^+ \leftarrow 1, v^- \leftarrow n$ 
15             until  $u = n + 1$ 
16              $u^- \leftarrow n, k^+ \leftarrow 1$ 
17         until  $k = n + 1$ 
18         .....
19     from  $k = n + 1$  do
20          $k^- \leftarrow 1, u^+ \leftarrow n$ 
21         from  $u = n + 1$  do
22              $v^+ \leftarrow n, u^- \leftarrow 1$ 
23             from  $v = n + 1$  do
24                  $v^- \leftarrow 1$ 
25                 if  $D^{k-1}[u,v] > (D^{k-1}[u,k] + D^{k-1}[k,v])$  then
26                      $D^k[u,v]^- \leftarrow D^{k-1}[u,k] + D^{k-1}[k,v]$ 
27                 else
28                      $D^k[u,v]^- \leftarrow D^{k-1}[u,v]$ 
29                 end-if
30             until  $v = 1$ 
31             until  $u = 1$ 
32         until  $k = 1$ 
33      $k^- \leftarrow 1, u^- \leftarrow 1, v^- \leftarrow 1$ 
34 end-FloydWarshall

```

mediário de estados que possa ser descartado sem comprometer a reversibilidade das iterações subsequentes. Tal impossibilidade refere-se à execução reversível direta, na qual cada etapa pode ser invertida sem recomputação.

No entanto, observamos que uma redução do espaço se torna possível quando se admite recomputar segmentos da execução a partir de estados intermediários armazenados (*checkpoints*) em combinação com estados momentaneamente descartados.

Por exemplo, para $n = 6$, originalmente mantemos as matrizes D^1 a D^5 em memória para computar D^6 . Em seguida, podemos zerar D^6 utilizando os valores de D^5 e assim sucessivamente até encontrar D^1 . Esta última pode então ser zerada com os dados de entrada, D^0 . Essa abordagem nos força a manter todas as matrizes anteriores até obter

D^n , consumindo espaço $\Theta(n^3)$. Porém, se utilizarmos D^3 como *checkpoint*, podemos, após computá-lo, zerar D^2 e D^1 , e desconsiderar essas matrizes – liberando espaço se feito de forma apropriada. O cálculo de D^4, D^5 e D^6 pode ser feito após essa fase. Após ser copiada para o resultado, D^6 pode ser zerada assim como D^5 e D^4 , lembrando que D^3 encontra-se armazenada. Todavia, para zerarmos D^3 de forma reversível torna-se necessária a recomputação de D^1 e de D^2 . Em seguida às recomputações, procedemos com a reversão de D^3, D^2 e, por fim, de D^1 . Nessa estratégia, a cada passo precisamos manter armazenadas somente as matrizes entre o *checkpoint* a ser computado e o anterior (ou D^0), bem como as matrizes correspondentes aos próprios *checkpoints*.

Assim, tomando como bloco básico uma matriz D^i de tamanho n^2 , a complexidade de espaço dependerá do número de *checkpoints* e do número de blocos do intervalo entre *checkpoints*. O cálculo para a minimização assintótica desta quantidade de blocos mantidos simultaneamente em memória utilizando *checkpoints* é simples considerando que um *checkpoint* está incluído no intervalo entre *checkpoints*. Se C é a quantidade de *checkpoints* e B é o intervalo entre *checkpoints*, as condições para determinar esses valores são:

$$\begin{aligned} &\text{minimize } (B - 1) + C \\ &\text{s.a.: } (B \cdot C) \geq n \end{aligned}$$

A solução é encontrada através da desigualdade das médias aritméticas e geométricas para números reais não negativos [31], que fornece o limite inferior de uma soma quando o produto dos termos é fixo:

$$\frac{B+C}{2} \geq \sqrt{B \cdot C} \Rightarrow B + C \geq 2\sqrt{n} \Rightarrow (B - 1) + C \geq 2\sqrt{n} - 1$$

Como os valores devem ser inteiros, e podemos desconsiderar os *checkpoints* dos intervalos, definimos:

- número de blocos entre *checkpoints*⁵:

$$B^* = B - 1 = \lceil \sqrt{n} \rceil - 1$$

- número de *checkpoints*:

$$C = \lceil \frac{n}{B} \rceil = \lceil \frac{n}{\lceil \sqrt{n} \rceil} \rceil$$

⁵Mesmo fora do modelo RAM transdicotômico, no qual é considerado uma primitiva que executa em tempo $O(1)$ com espaço $O(1)$, o cálculo de $O(\sqrt{n})$ pode ser computado reversivelmente de forma iterativa em tempo $O(\sqrt{n})$ no início do procedimento, de forma a não afetar a complexidade do Algoritmo 7. A divisão de n por $\sqrt{n}$ é reversível uma vez que opera sobre valores que não se perdem durante a computação.

Note que

$$C \cdot B = \left\lceil \frac{n}{\lceil \sqrt{n} \rceil} \right\rceil \cdot \lceil \sqrt{n} \rceil \geq 2\sqrt{n}, \text{ para todo } n \geq 3.$$

Além disso,

$$C = \left\lceil \frac{n}{\lceil \sqrt{n} \rceil} \right\rceil \leq \lceil \sqrt{n} \rceil.$$

Logo,

$$2 \cdot \sqrt{n} - 1 \leq (B^* + C) \leq 2 \cdot \lceil \sqrt{n} \rceil - 1$$

Portanto, a complexidade de espaço da abordagem previamente descrita é $\Theta(n^{2.5})$, pois o espaço auxiliar utilizado possui tamanho:

$$(B^* + C) \cdot n^2 \leq 2\lceil \sqrt{n} \rceil \cdot n^2 - n^2$$

Uma vez que os blocos são recomputados e descomputados apenas mais uma vez em comparação com o Algoritmo 6, a complexidade de tempo se mantém, ou seja, $\Theta(n^3)$. A Figura 27 descreve o fluxo de computações, recomputações e reversões para o exemplo dado, com $n = 6$, para o qual $B = 3$ e $B^* = C = 2$.

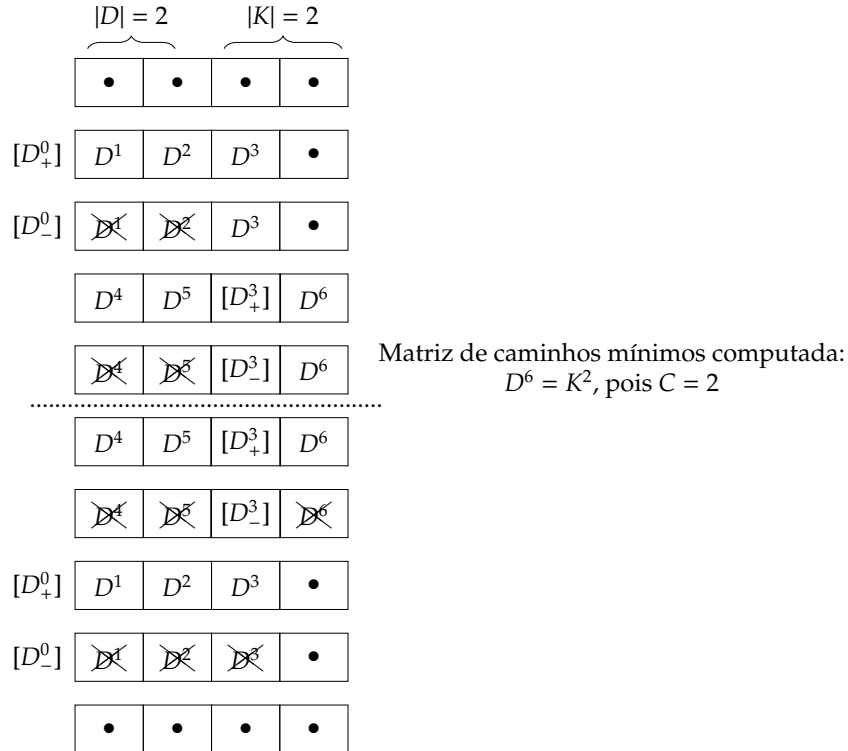


Figura 27: Fluxo de computações, recomputações e reversões do algoritmo reversível Floyd-Warshall com checkpoints para $n = 6$. Entre colchetes está a matriz original D^i que permite a computação, recomputação ($[D^i_+]$) ou reversão ($[D^i_-]$) em cada caso.

O que descrevemos está codificado no Algoritmo 7. A variável *phase* seleciona a execução direta ou reversa. O vetor D , agora de tamanho $B^* = \lceil \sqrt{n} \rceil - 1$, recebe os blocos entre *checkpoints*, e o vetor K , de tamanho $C = \lceil \frac{n}{\sqrt{n}} \rceil$, os *checkpoints*. Sequencialmente, os blocos entre os *checkpoints* são computados, selecionando a matriz de origem do cálculo, a cada caso: esta pode ser a matriz original D^0 , a matriz anterior do intervalo ou o último *checkpoint* computado. Em seguida, é computado o *checkpoint* e os blocos do intervalo anterior são revertidos. O procedimento é repetido até a computação do último *checkpoint* em K^C , que conterá a matriz dos caminhos mínimos de G . A reversão ocorre em *phase* = 2, com a recomputação dos blocos entre os *checkpoints* seguida da reversão dos *checkpoints* e dos blocos do intervalo entre o *checkpoint* e seu anterior, até a reversão de D^1 a partir de D^0 .

Note que os cálculos, reversões e recálculos de matrizes utilizam k juntamente com u e v para acesso às distâncias entre os pares de vértices (u,v) , (u,k) e (k,v) , de acordo com a regra de recorrência do problema, independentemente de os vetores D e K da solução, através dos índices b e cp , acessarem um número menor de matrizes. Tal fato revela mesmo a alma da solução: manter o acesso correto aos valores internos das n possíveis matrizes mesmo que estejam mapeadas no espaço reduzido de $2\sqrt{n}$ matrizes.

A solução que mostramos acima parece constituir um compromisso assintoticamente ótimo de espaço *versus* tempo para o algoritmo reversível. Nosso resultado obteve consumo de espaço assintoticamente inferior ao utilizado pela versão reversível do Floyd-Warshall dada por Demaine *et al.* [12] e Frank and Knight Jr. [23]: consumimos espaço $\Theta(n^{2.5})$, enquanto eles utilizavam $\Theta(n^3)$. Além disso, nosso algoritmo é mais eficiente com relação ao tempo de execução quando comparado com o método reversível proposto por Demaine *et al.* [12] para computação de caminhos mínimos baseada no algoritmo de multiplicação de matrizes $(\min,+)$: solucionamos o problema em tempo $\Theta(n^3)$ enquanto eles solucionavam o problema em tempo $O(n^3 \log n)$.

Nosso método é o primeiro algoritmo reversível proposto na literatura para computar os custos dos caminhos mínimos entre todos os pares de vértices de um grafo ponderado sem ciclos negativos, executando em tempo $O(n^3)$ e consumindo espaço $o(n^3)$. Desse modo, mostramos que é possível preservar a complexidade de tempo do algoritmo clássico não reversível ao custo de apenas um aumento sublinear na complexidade de espaço, mais especificamente por um fator de $\sqrt{n}$.

A abordagem em blocos proposta neste trabalho estabelece um único nível de *checkpoints* que pode conter uma quantidade de blocos maior ou igual a 1. Este constitui

o caso base ($g = 1$) de uma estratégia parametrizável de *checkpoints* multinível, dado que é teoricamente possível estender o modelo para uma profundidade constante $g > 1$, onde $g \ll \log n$, o que permitiria comprimir o espaço de armazenamento para patamares ainda menores do que $\sqrt{n}$ retendo a complexidade assintótica de tempo em $O(n^3)$ devido ao acréscimo de apenas um fator multiplicativo constante.

O modelo clássico de Bennett [7], que adaptou as regras do jogo *pebble game* [56] às regras da reversibilidade, surge como limite superior da generalização que citamos, ocorrendo estritamente quando a árvore atinge a sua profundidade máxima ($g \approx \log n$) e os blocos colapsam para a unidade ($B^* = C_1^* = \dots = C_g^* = 1$). O projeto do algoritmo general e sua comparação com o *pebble game* constituem a extensão natural mapeada para trabalhos futuros – ver página 103.

Algoritmo 7: *FloydWarshall* reversível com checkpoints

```

1  FloydWarshall
2   $cp \leftarrow 1, C \leftarrow \lceil \frac{n}{\sqrt{n}} \rceil, B \leftarrow \lceil \sqrt{n} \rceil, phase \leftarrow 1$ 
3  from  $phase = 1$  do
4      from  $cp = (phase - 1) \cdot C + 1$  do
5          if  $phase = 2$  then  $cp^- \leftarrow 1$  end-if
6           $b^+ \leftarrow 1, u^+ \leftarrow 1, v^+ \leftarrow 1, k^+ \leftarrow (cp - 1) \cdot B + 1$ 
7          from  $b = 1$  do
8              from  $u = 1$  do
9                  from  $v = 1$  do
10                     if  $b = 1$  and  $cp = 1$  then
11                          $D^1[u, v]^+ \leftarrow \min \{ D^0[u, v], D^0[u, 1] + D^0[1, v] \}$ 
12                     end-if
13                     if  $b > 1$  then
14                          $D^b[u, v]^+ \leftarrow \min \{ D^{b-1}[u, v], D^{b-1}[u, k] + D^{b-1}[k, v] \}$ 
15                     end-if
16                     if  $b = 1$  and  $cp > 1$  then
17                          $D^b[u, v]^+ \leftarrow \min \{ K^{cp-1}[u, v], K^{cp-1}[u, k] + K^{cp-1}[k, v] \}$ 
18                     end-if
19                      $v^+ \leftarrow 1$ 
20                 until  $v = n + 1$ 
21                  $u^+ \leftarrow 1, v^- \leftarrow n$ 
22             until  $u = n + 1$ 
23              $u^- \leftarrow n, b^+ \leftarrow 1, k^+ \leftarrow 1$ 
24         until  $(b = B) \text{ or } (k = n)$ 
25         .....
26     if  $phase = 1$  then
27          $K^{cp}[u, v]^+ \leftarrow \min \{ D^{b-1}[u, v], D^{b-1}[u, k] + D^{b-1}[k, v] \}, \forall u, v \in V(G)$ 
28     else
29          $K^{cp}[u, v]^- \leftarrow \min \{ D^{b-1}[u, v], D^{b-1}[u, k] + D^{b-1}[k, v] \}, \forall u, v \in V(G)$ 
30     end-if
31     .....
32     from  $(b = B) \text{ or } (k = n)$  do    % Note que o teste equivale a " $b = k - (cp - 1) \cdot B$ ".
33          $u^+ \leftarrow n, b^- \leftarrow 1, k^- \leftarrow 1$ 
34         from  $u = n + 1$  do
35              $v^+ \leftarrow n, u^- \leftarrow 1$ 
36             from  $v = n + 1$  do
37                  $v^- \leftarrow 1$ 
38                 if  $b = 1$  and  $cp > 1$  then
39                      $D^b[u, v]^- \leftarrow \min \{ K^{cp-1}[u, v], K^{cp-1}[u, k] + K^{cp-1}[k, v] \}$ 
40                 end-if
41                 if  $b > 1$  and then
42                      $D^b[u, v]^- \leftarrow \min \{ D^{b-1}[u, v], D^{b-1}[u, k] + D^{b-1}[k, v] \}$ 
43                 end-if
44                 if  $b = 1$  and  $cp = 1$  then
45                      $D^1[u, v]^- \leftarrow \min \{ D^0[u, v], D^0[u, 1] + D^0[1, v] \}$ 
46                 end-if
47             until  $v = 1$ 
48         until  $u = 1$ 
49     until  $b = 1$ 
50      $b^- \leftarrow 1, u^- \leftarrow 1, v^- \leftarrow 1, k^- \leftarrow (cp - 1) \cdot B + 1$ 
51     if  $phase = 1$  then  $cp^+ \leftarrow 1$  end-if
52     until  $cp = C + 1 - (phase - 1) \cdot C$ 
53      $phase^+ \leftarrow 1$ 
54     % Note que, após a fase 1,  $K^C$  é a matriz dos caminhos mínimos de G.
55 until  $phase = 3$ 
56  $cp^- \leftarrow 1, C^- \leftarrow \lceil \frac{n}{\sqrt{n}} \rceil, B^- \leftarrow \lceil \sqrt{n} \rceil, phase^- \leftarrow 3$ 
57 end-FloydWarshall

```

7.3 Radix Sort

7.3.1 O algoritmo clássico

Radix Sort é um método de ordenação de inteiros que não efetua comparação de elementos. Cada iteração considera apenas um dígito dos elementos por vez, começando do menos significativo para o mais significativo. Do primeiro ao último, cada um dos n elementos é removido do vetor V e destinado temporariamente à fila determinada pelo valor do dígito em análise. Há uma fila para cada um dos 10 símbolos possíveis da base decimal (Figura 28.a). A recomposição de V ocorre, no mesmo sentido, pelo esvaziamento completo de cada fila, da 0 à 9 (Figura 28.b). O procedimento é repetido o mesmo número de vezes que a quantidade d de dígitos do maior elemento de V , e então V estará ordenado. O Radix Sort (Algoritmo 8) tem complexidade de tempo e de espaço igual a $O(d \cdot n)$. No entanto, o Radix Sort pode ser refinado para performar em tempo e espaço $O(n^*)$, onde n^* é o número de bits para codificar a entrada. Note que n^* pode ser muito menor que $d \cdot n$ se os inteiros da entrada tiverem tamanhos distintos.

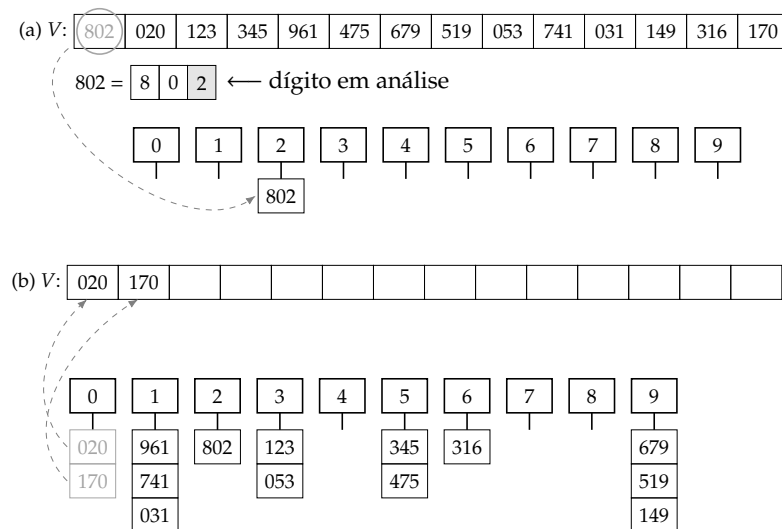


Figura 28: Ordenação pelo processo repetido de esvaziamento (a) e recomposição (b) do vetor usando filas (*queues* ou *buckets*), que recebem os elementos com base no valor do dígito significativo analisado na iteração.

O método é dito *estável* por preservar a ordem relativa de elementos iguais na solução final. O uso de filas é fundamental para a estabilidade, pois garante que o primeiro elemento originário do vetor com um dado valor retornará a ele antes de qualquer outro elemento com o mesmo valor.

Algoritmo 8: Radix Sort clássico

```

1  Radix( $V, d, n$ )
2  para todo  $i_d \in [1, d]$ 
3      para todo  $i \in [1, n]$ 
4           $c \leftarrow i_d$ -ésimo dígito menos significativo de  $V[i]$ 
5          move o elemento  $V[i]$  para a  $c$ -ésima fila
6      fim-para
7       $i \leftarrow 1$ 
8      para todo  $c \in [0, 9]$ 
9          enquanto a  $c$ -ésima fila não estiver vazia
10             move o elemento da  $c$ -ésima fila para  $V[i]$ 
11              $i \leftarrow i + 1$ 
12          fim-enquanto
13      fim-para
14  fim-para
15  fim-Radix

```

7.3.2 Dificuldades enfrentadas na reversibilização

Em suas versões clássicas, os algoritmos de ordenação não são bijetivos: a transformação global que operam é de muitos-para-um, ou seja, diferentes disposições iniciais para os mesmos elementos convergem para uma única versão final ordenada. Dito de outra forma, o problema não é determinístico de trás para frente (*backward deterministic*). Encontrar um algoritmo reversível passa forçosamente por identificar as ocorrências de perda de informação para solucionar essa indeterminação reversa. Aqui, a informação que se perde é, a cada movimentação, a posição anterior ocupada pelo elemento.

Consideremos, a princípio, que todos os n elementos possuem a mesma quantidade d de dígitos, de forma que o i_d -ésimo dígito significativo (o menor sendo $i_d = 1$) do i -ésimo elemento possa ser obtido através de uma função f , tal que $r \leftarrow f(V[i], i_d)$, onde r representa a fila de destino do elemento na iteração que analisa o dígito i_d . No exemplo exposto na Figura 28, para $i = 1$ e $i_d = 1$, temos $r = 2$, de modo que o elemento 802 é movido para a fila 2, bastando que se realize uma troca atômica entre o conteúdo da primeira posição de V e a posição de inserção (*tail*) da fila r (que assumimos zerada⁶).

Na versão clássica, após a movimentação, r pode ser sobrescrita sem problemas em seu uso seguinte na determinação da fila de destino do próximo elemento; aqui, r deve ser revertida a zero antes de ser reutilizada⁷, para que não seja fonte de gasto

⁶Lembremos que a máquina deve disponibilizar memória inicialmente zerada por conta dos princípios da computação reversível [68].

⁷Mesmo que no caso em questão a reversão imediata seja imperiosa, Axelsen [2] ilustra muito bem como a prática de reverter uma variável assim que possível evita o acúmulo da quantidade de lixo (*garbage*), que pode impactar na complexidade de espaço de um algoritmo.

energético. Instala-se, então, um impasse: o valor necessário para reverter localmente a variável que determina a fila r de destino de cada elemento é o i_d -ésimo dígito do último elemento inserido na fila r , e as regras da reversibilidade não permitem operações do tipo $x^- \leftarrow exp(x)$, i.e., operações nas quais a expressão que subtrai a variável que se pretende modificar contém essa mesma variável⁸ [51, 2].

7.3.3 Sobre a bijetivação do problema e a mecânica da solução

A nossa solução para a reversão de r é o salvamento de seu valor, que efetuamos na pilha do sistema através do comando $push(r)$: este realiza uma troca atômica entre r e o topo (zerado) da pilha e incrementa o ponteiro referente ao topo. Isto posto, resolvemos dois problemas com um só golpe: revertermos a variável r e salvamos o valor que torna a solução bijetiva, pois *a informação da posição original do elemento se dilui na informação da fila a qual foi destinado o elemento em conjunto com a posição desta última informação na pilha*. Tal diluição reafirma a postura que a reversibilidade pede que adotemos em relação às listas: que as tratemos como pilhas. Uma vez que utilizamos a pilha do sistema e para não gerar confusão, daqui por diante trataremos as filas por listas.

Ao final de um esvaziamento, o índice de acesso a V se encontra no valor máximo igual a n . Então, ao invés de remontarmos o vetor da primeira à última posição pelo consumo da lista 0 à 9, procedemos no sentido contrário: i.e., da última para a primeira posição do vetor, consumindo as listas no regime último-elemento-que-entra-primeiro-que-sai (*Last In, First Out* – LIFO), ou seja, como pilhas, desde a 9 até a 0. Uma vez ordenado após d iterações, V , agora rebatizado de V_{in} , é copiado para V_{out} e passa a ser revertido. A cada início de iteração da fase reversa, V_{in} se encontra ordenado pelo dígito i_d que comanda a iteração, de modo que os elementos, do primeiro ao último, são devolvidos às suas listas respectivas, também da primeira à última, da 0 à 9. Movimentado todo o vetor, encontramos-nos outra vez no ponto em que o vetor será reconstruído em sentido contrário, do último ao primeiro elemento. Desta vez, porém, o consumo das listas não segue uma ordem pré-estabelecida, mas é ditada pela informação salva na pilha do sistema. Esse valor, lembremos, informa qual lista deve fornecer o elemento a ocupar novamente uma posição em V_{in} . A natureza de pilha é perfeita para oferecer, na fase reversa, os valores acumulados na fase direta do algoritmo. Após mais d iterações, toda a informação da pilha do sistema terá sido

⁸ $r^- \leftarrow f(\text{fila } r, i_d)$ destrói o valor prévio de r . A expressão $r^+ \leftarrow f(\text{fila } r, i_d)$ não funciona como sua inversa, pois a fila 0 seria endereçada sempre.

utilizada; a mesma se encontrará vazia. Da mesma forma estarão zeradas as listas e os índices. V_{in} haverá recuperado sua disposição inicial de elementos.

7.3.4 A questão do espaço – não sem a do tempo

Uma vez que é possível reversibilizar qualquer algoritmo através da utilização de espaço suplementar bennett1973logical, um desafio é fazer com que este último não cresça demasiadamente. O mais usual é que a complexidade de espaço acabe por alcançar a de tempo ou mesmo a ultrapasse. Aqui, nossa tentativa será a de manter a complexidade de espaço da versão clássica, com um detalhe.

No caso real, não é certo, sendo em verdade muito raro, que os elementos do vetor a ser ordenado conttenham a mesma quantidade de dígitos. Isto posto, essa possível diferença acarretará a movimentação desnecessária de elementos para ordenar zeros à esquerda, caso o tamanho do elemento seja dimensionado a partir do maior, ou para ordenar os elementos que já não possuïrem tal dígito. Para otimizar esse cenário, o algoritmo deve:

- (a) quanto ao espaço: codificar os elementos em tamanhos variáveis e diretamente proporcionais à sua quantidade de dígitos; e
- (b) quanto ao tempo: desconsiderar, nas próximas iterações de ordenação, os elementos cuja totalidade de dígitos já estiver sido analisada, i.e., os já ordenados.
- (c) ainda quanto ao tempo: embora os dígitos devam ser tratados individualmente, a movimentação de cada elemento entre o vetor e as filas deve ser executada, como na versão clássica, em $O(1)$.

A seguir comentamos nossa solução de acordo com essas diretrizes, bem como suas estruturas auxiliares de suporte e de como a mecânica se insere no algoritmo final.

Representação da entrada e da saída. Cada elemento é representado como uma sequência explícita de dígitos decimais precedida pelo comprimento desta sequência. Por exemplo, o elemento 468 aparece como a *substring* 3468; a unidade básica é o inteiro, então um elemento de 11 dígitos seria precedido de 11. Se o i -ésimo elemento possui s_i dígitos, o tamanho total da entrada, em caracteres, é:

$$n^* = \sum_{i=1}^n (s_i + 1)$$

Note que n^* , como definido acima, é da ordem do número de bits para codificar a entrada, V_{in} , e esta é também a estrutura adotada para V_{out} . Dois índices independentes são utilizados: i para elementos e i^* para dígitos. As análises assintóticas usam n^* em vez de n . É assumido o modelo RAM transdicitômico, no qual índices e endereços se encaixam em palavras de tamanho $\Theta(\log n^*)$.

A transformação interna da representação dos elementos. Para que satisfaçamos as condições acima expostas nossa solução foi a de operar internamente com um vetor de n posições, onde cada uma armazena o índice ou o ponteiro para a primeira posição do elemento em V_{in} . Por um lado, o ponteiro permite sua movimentação em $O(1)$; por outro, a estrutura dos dados em V_{in} facilita o acesso individual dos dígitos. A Figura 29 ilustra a transformação.

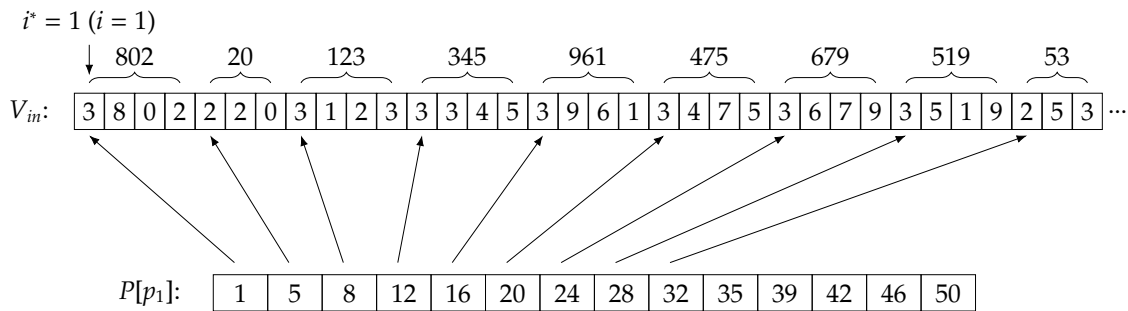


Figura 29: A transformação da representação de entrada dos elementos V_{in} no vetor de ponteiros $P[p_1]$

As listas e seu tratamento. As listas são acomodadas sobre espaços reservados sequencialmente sobre um vetor de comprimento também igual a n , uma vez que por elas transitarão os ponteiros, e não os elementos em sua representação de entrada. Nesse vetor, as primeiras posições acomodam os elementos da lista 0, se existirem, e assim por diante até os da lista 9.

A reserva de espaço sobre o vetor de listas para cada uma das listas depende da realização prévia, a cada iteração: da contagem da quantidade total de ponteiros correspondente aos elementos que terão como destino cada lista r_i , e, a partir desses valores, do cálculo da posição inicial de inserção de cada lista r_i sobre o vetor de listas. Assim, se em seu início, a primeira iteração de ordenação ($i_d = 1$) se depara com o elemento 802 (formato original: 3802) e a posição de inserção da fila 2 é igual a 6, é realizada a troca atômica entre o valor (igual a 1) do ponteiro correspondente ao elemento 1, e a posição 6 do vetor de listas, com o ponto de inserção da lista 2 atualizado para 7. O movimento reverso não é mais complicado: determinada a lista

r para consumo a partir do valor salvo em pilha, o valor do ponteiro imediatamente anterior à posição de inserção da lista r é movido para a posição anterior à i -ésima do vetor interno.

Alternando o vetor com as listas. A distribuição dos intervalos correspondentes às listas sobre um vetor de n posições tem como consequência que, uma vez tendo recebido todos os elementos e considerados apenas os elementos não fixados, é idêntico ao vetor: a) ao qual dará origem por remontagem, em uma ordenação na fase direta; e b) que lhe originou, em uma desordenação na fase reversa.

Isto posto, economizamos as movimentações de simples retorno dos elementos das listas para o vetor, na fase direta, e de envio dos elementos para as listas, na fase reversa, pois as ordenações (e desordenações) por dígito podem ser concluídas com movimentações realizadas em um único sentido. Os dois vetores internos, o que representa a ordem dos elementos a cada passo e o que acomoda as filas, cabem dentro do vetor bidimensional P , de modo que, inicialmente, $P[1]$ recebe a transformação de V_{in} em ponteiros. As variáveis p_1 e p_2 , inicializadas respectivamente em 1 e 2, são utilizadas como chaves que se invertem para determinar, a cada iteração, seja da fase direta ou reversa, qual índice do vetor acessa os elementos a ordenar ou desordenar e qual acessa as listas. Por exemplo, na ordenação do primeiro dígito, $P[1]$ funciona como o vetor a ser ordenado e $P[2]$ como as listas. Ao final da movimentação, as variáveis são invertidas através de uma troca atômica ($p_1 \leftrightarrow p_2$, que é sua própria inversa), de modo que $P[2]$ passa a representar o vetor e $P[1]$ as listas, e assim por diante.

Fixando elementos já ordenados. Após ter todos os seus dígitos analisados, um elemento pode ser esquecido. Na prática, uma iteração extra é necessária para destiná-lo à sua posição definitiva em $P[1]$ ou $P[2]$, que ocorre quando sua quantidade de dígitos é menor do que o índice do dígito que está sendo analisado ($s_i < d_i$). Como os elementos já ordenados são "decantados" progressivamente nas primeiras posições de V_{in} , é preciso reservar a primeira fila para receber, a cada iteração, os elementos ainda não fixados e que serão desconsiderados pelas iterações posteriores. Então:

- os elementos para os quais $s_i < i_d$ são destinados à lista 0; para todos os demais casos, será somado 1 ao número da lista ao qual inicialmente o elemento seria destinado (e calculada como no item anterior), de modo que $P[p_2]$ hospeda até 11 listas, da 0 à 10; e
- cada iteração tem a seu dispor o somatório dos elementos ordenados nas iterações anteriores (*Fixed*). Essa quantidade determina os valor iniciais do índice i a cada

iteração, de modo que os elementos anteriores são desconsiderados no novo ciclo. Na fase reversa, revela-se o oposto: cada iteração considera uma quantidade de elementos cada vez maior.

Obtenção do resultado. Ao longo do processo de ordenação, os elementos fixados são distribuídos entre $P[1]$ e $P[2]$ de acordo com a quantidade de dígitos (ímpar ou par, respectivamente) dos elementos. Assim, o ponteiro final de cada elemento para acessar V_{in} com vistas à construção de V_{out} deve ser obtido com a soma de $P[1]$ a $P[2]$, posição a posição: $P[1, i] + P[2, i], \forall i^* \in [1, \dots, n]$. Isso é possível porque, para uma mesma posição i , um dos termos é sempre igual a zero. A fase reversa é completada com a reversão de $P[1]$ e dos índices.

Estabilidade. A estabilidade é preservada porque os elementos são inseridos nas listas mantendo a ordem de leitura e removidos de forma consistente com essa ordem.

7.3.5 Pseudocódigo e análise da complexidade

A estrutura geral da solução é apresentada no Algoritmo 9. O pseudocódigo principal e suas subrotinas compõem os Algoritmos 10 e 11.

Detalhes de implementação. O algoritmo recebe como entrada V_{in} , V_{out} (vazio), n , n^* e d . *InitP* alimenta $P[1]$ com os ponteiros para os elementos em V_{in} . A quantidade de elementos destinados a cada lista e a posição de inserção inicial de cada lista em $P[p_2]$ ficam a cargo dos vetores de 11 posições L_{size} e L_{tail} , respectivamente, que são revertidos ao final de cada iteração. *Fixed*, vetor de comprimento $d + 1$, é responsável por informar a quantidade de elementos fixados em uma determinada iteração i_d . p_1 e p_2 , iniciados em 1 e 2, respectivamente, invertem seus valores para que $P[p_1]$ sempre referencie o vetor ordenado de ponteiros e $P[p_2]$ referencie as listas.

A função que retorna o valor do dígito em análise para o elemento corrente (*WhichList*) e a que realiza a contagem dos elementos de cada lista (*ComputeListsSizes*), bem como suas reversas, recebem como parâmetro o vetor no qual devem procurar a informação ou sobre o qual devem operar, em cada caso: se em p_1 ou em p_2 .

Como poupamos movimentações desnecessárias, os pontos de inserção das listas não são decrementados antes da reversão da contagem de elementos, de modo que a função que realiza o cálculo e a reversão dos L_{tails} , *ComputeListsTails* e sua inversa *ComputeListsTails*⁻¹, são parametrizadas com vistas à determinação do procedimento correto a ser observado em cada caso.

Por economia e visando à simplificação, tratamos a maioria das variáveis como globais. Anotamos algumas pré e pós-condições a serem observadas por comandos [19, 33, 14, 29] para facilitar o acompanhamento de estados pelo leitor. A Figura 30 ilustra as estruturas de dados em jogo e a movimentação de um ponteiro para ordenação, com a escolha de lista e armazenamento de informação para reversão.

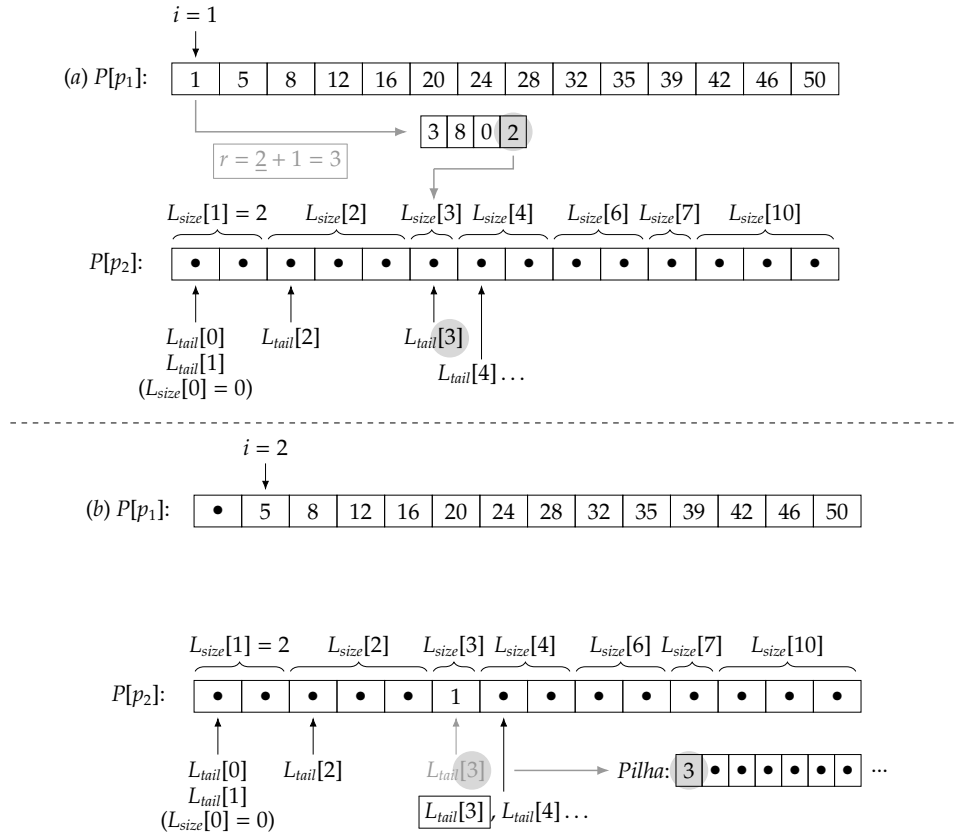


Figura 30: Movimento do primeiro ponteiro para sua lista correspondente, na iteração $i_d = 1$. Os ponteiros para os elementos a serem ordenados $P[p_1]$ e as listas $P[p_2]$ podem ser visualizados, juntamente com informações auxiliares sobre contagem de elementos, pontos inserção das listas e índices de percurso das estruturas. Os destaques em cinza indicam o que está em jogo na escolha da lista de destino para um elemento e as informações que são inseridas para permitir a reversão – (a) e (b). [•] indica uma posição zerada.

Algoritmo 9: Radix Sort reversível: estrutura geral

```

1   $\text{RevRadix}(V_{in}, n^*, n, d, V_{out})$ 
2  inicializa  $p_1 = 1, p_2 = 2$ 
3  computa o vetor de ponteiros  $P[p_1]$  a partir de  $V_{in}$ 
4  para todo dígito  $i_d$ , do menos ao mais significativo
5      posiciona o índice no 1º ponteiro não fixado
6      computa o tamanho e o ponto de inserção de cada lista  $r$  em  $P[p_2]$  (da 0 à 10)
7      para todo ponteiro em  $P[p_1]$ , do 1º não fixado ao último
8          determina a lista  $r$  a mover o ponteiro a partir da consulta a  $V_{in}$ 
9          se  $r = 0$  então fixa o ponteiro a partir da iteração  $i_d + 1$ 
10         move o ponteiro para a fila  $r$ 
11         empilha  $r$ 
12     fim-para
13     descomputa o tamanho e o ponto de inserção cada lista  $r$  em  $P[p_2]$  (da 10 à 0)
14     soma n° de fixações da iteração  $i_d$  nos da iteração  $i_d + 1$ 
15     troca atonicamente os valores de  $p_1$  e  $p_2$ 
16 fim-para
17 .....
18 computa  $V_{out}$  a partir da soma  $P[p_1][i] + P[p_2][i], \forall i \in [1, n]$ 
19 .....
20 para todo dígito  $i_d$ , do mais ao menos significativo faça
21     troca atonicamente os valores de  $p_1$  e  $p_2$ 
22     reverte o n° de fixações da iteração  $i_d + 1$  a partir do valor ref. à iteração  $i_d$ 
23     posiciona o índice no 1º ponteiro não fixado
24     computa o tamanho e o ponto de inserção de cada lista  $r$  em  $P[p_2]$  (da 0 à 10)
25     para todo elemento de  $V_{in}$ , do último ao 1º não fixado faça
26         desempilha  $r$ 
27         move o ponteiro do fim da lista  $r$  para a última posição livre de  $V_{in}$ 
28         se  $r = 0$  então desfaz a fixação do ponteiro a partir da iteração  $i_d - 1$ 
29         reverte  $r$  a partir das informações do ponteiro movido
30     fim-para
31     descomputa o ponto de inserção e o comprimento de cada lista  $r$  (da 10 à 0)
32 fim-para
33 descomputa o vetor de ponteiros  $P[p_1]$  a partir de  $V_{in}$ 
34 reverte  $p_1, p_2, i, i^*$ 
35 fim-RevRadix
  
```

Algoritmo 10: Radix Sort reversível: principal

```

1  RevRadix
2   $\{i^*, i, i_d, r, p_1, p_2, P[2, n], L_{size}[11], L_{tail}[11], Fixed[d + 1], V_{out}[n^*] = 0\}$ 
3   $\underline{i} \leftarrow 1, \underline{i}^* \leftarrow 1, \underline{i}_d \leftarrow 1, \underline{p}_1 \leftarrow 1, \underline{p}_2 \leftarrow 2$ 
4   $InitP()$ 
5  from  $i_d = 1$  do
6       $i^+ \leftarrow Fixed[i_d]$ 
7       $ComputeListsSizes(p_1)$ 
8       $ComputeListsTails(1)$ 
9       $\{i = Fixed[i_d] + 1\}$ 
10     from  $i = Fixed[i_d] + 1$  do
11          $\underline{r} \leftarrow WhichList(p_1)$ 
12         if  $r = 0$  then  $Fixed[i_d + 1]^+ \leftarrow 1$  end-if
13          $P[p_1, i] \leftrightarrow P[p_2, L_{tail}[r]]$ 
14          $L_{tail}[r]^+ \leftarrow 1, i^+ \leftarrow 1$ 
15          $push(r)$ 
16     until  $i = n + 1$ 
17      $i^- \leftarrow (n - Fixed[i_d]) \{i = Fixed[i_d] + 1\}$ 
18      $ComputeListsTails^{-1}(0)$ 
19      $ComputeListsSizes^{-1}(p_2)$ 
20      $\{i = Fixed[i_d] + 1\}$ 
21      $i^- \leftarrow Fixed[i_d] \{i = 1\}$ 
22      $Fixed[i_d + 1]^+ \leftarrow Fixed[i_d]$ 
23      $i_d^+ \leftarrow 1$ 
24      $p_1 \leftrightarrow p_2$ 
25 until  $i_d = d + 1$ 
26 .....
27  $BuildV_{out}FromP()$ 
28 .....
29 from  $i_d = d + 1$  do
30      $p_1 \leftrightarrow p_2$ 
31      $i_d^- \leftarrow 1$ 
32      $Fixed[i_d + 1]^- \leftarrow Fixed[i_d]$ 
33      $i^+ \leftarrow Fixed[i_d]$ 
34      $ComputeListsSizes(p_2)$ 
35      $ComputeListsTails(0)$ 
36      $\{i = Fixed[i_d] + 1\}$ 
37      $i^+ \leftarrow n - Fixed[i_d] \{i = n + 1\}$ 
38     from  $i = n + 1$  do
39          $pop(r)$ 
40          $L_{tail}[r]^- \leftarrow 1, i^- \leftarrow 1$ 
41          $P[p_1, i] \leftrightarrow P[p_2, L_{tail}[r]]$ 
42         if  $r = 0$  then  $Fixed[i_d + 1]^- \leftarrow 1$  end-if
43          $\underline{r}^- \leftarrow WhichList(p_1)$ 
44     until  $i = Fixed[i_d] + 1$ 
45      $ComputeListsTails^{-1}(1)$ 
46      $ComputeListsSizes^{-1}(p_1)$ 
47      $\{i = Fixed[i_d] + 1\}$ 
48      $i^- \leftarrow Fixed[i_d]$ 
49 until  $i_d = 1$ 
50  $InitP^{-1}()$ 
51  $i^- \leftarrow 1, i^* \leftarrow 1, i_d^- \leftarrow 1, \underline{p}_1^- \leftarrow 1, \underline{p}_2^- \leftarrow 2$ 
52 end-RevRadix

```

```

1  InitP
2  {  $i = 1$  }
3   $P[p_1, 1] \leftarrow 1$ 
4   $i^+ \leftarrow 1$ 
5  from  $i = 2$  do
6     $P[p_1, i] \leftarrow P[p_1, i-1] + V_{in}[P[p_1, i-1]] + 1$ 
7     $i^+ \leftarrow 1$ 
8  until  $i = n + 1$ 
9   $i^- \leftarrow n$ 
10 {  $i = 1$  }
11 end-InitP

```

```

1  ComputeListsSizes(p)
2  from  $i = \text{Fixed}[i_d] + 1$ 
3     $r \leftarrow \text{WhichList}(p)$ 
4     $L_{size}[r]^+ \leftarrow 1$ 
5     $r^- \leftarrow \text{WhichList}(p)$ 
6     $i^+ \leftarrow 1$ 
7  until  $i = n + 1$ 
8   $i^- \leftarrow n - \text{Fixed}[i_d]$ 
9  {  $i = \text{Fixed}[i_d] + 1$  }
10 end-ComputeListsSizes

```

```

1  ComputeListsTails(previous)
2  {  $r = 0$  }
3   $L_{tail}[0] \leftarrow \text{Fixed}[i_d] + 1$ 
4  if  $\text{previous} \oplus 1$  then
5     $L_{tail}[0]^+ \leftarrow L_{size}[0]$ 
6  end-if
7  from  $r = 0$  do
8     $r^+ \leftarrow 1$ 
9     $L_{tail}[r] \leftarrow L_{tail}[r-1] +$ 
10      $L_{size}[r - \text{previous}]$ 
11 until  $r = 10$ 
12  $r^- \leftarrow 10$ 
13 end-ComputeListsTails

```

```

1  WhichList(p)
2  {  $c = 0$  }
3  if  $V_{in}[P[p, i]] \geq i_d$  then
4     $c \leftarrow V_{in}[P[p, i]] + V_{in}[P[p, i]] - (i_d - 1) + 1$ 
5  end-if
6  return  $c$ 
7  end-WhichList

```

```

1  InitP-1
2  {  $i = 1$  }
3   $i^+ \leftarrow n$ 
4  from  $i = n + 1$  do
5     $i^- \leftarrow 1$ 
6     $P[p_1, i]^- \leftarrow P[p_1, i-1] + V_{in}[P[p_1, i-1]] + 1$ 
7  until  $i = 2$ 
8   $i^- \leftarrow 1$ 
9   $P[p_1, 1]^- \leftarrow 1$ 
10 {  $i = 1$  }
11 end-InitP-1

```

```

1  ComputeListsSizes-1(p)
2  {  $i = \text{Fixed}[i_d] + 1$  }
3   $i^+ \leftarrow n - \text{Fixed}[i_d]$ 
4  from  $i = n + 1$  do
5     $i^- \leftarrow 1$ 
6     $r \leftarrow \text{WhichList}(p)$ 
7     $L_{size}[r]^- \leftarrow 1$ 
8     $r^- \leftarrow \text{WhichList}(p)$ 
9  until  $i = \text{Fixed}[i_d] + 1$ 
10 end-ComputeListsSizes-1

```

```

1  ComputeListsTails-1(previous)
2  {  $r = 0$  }
3   $r^- \leftarrow 10$ 
4  from  $r = 10$  do
5     $L_{tail}[r]^- \leftarrow L_{tail}[r-1] +$ 
6      $L_{size}[r - \text{previous}]$ 
7     $r^- \leftarrow 1$ 
8  until  $r = 0$ 
9  if  $\text{previous} \oplus 1$  then
10     $L_{tail}[0]^- \leftarrow L_{size}[0]$ 
11 end-if
12  $L_{tail}[0]^- \leftarrow \text{Fixed}[i_d] + 1$ 
13 end-ComputeListsTails-1

```

```

1  BuildVoutFromP
2  {i = 1, i* = 1, m = 0}
3  from i = 1 do
4      from m = 0 do
5          Vout[i*] ← Vin[P[p1,i] + P[p2,i] + m]
6          m+ ← 1, i*+ ← 1
7          until m = Vin[P[p1,i] + P[p2,i]] + 1
8          m- ← Vin[P[p1,i] + P[p2,i]] + 1
9          i+ ← 1
10 until i = n + 1 {i* = n* + 1}
11 i- ← n, i*- ← n*
12 {i* = 1, i = 1}
13 end-BuildVoutFromP

```

Algoritmo 11: Rotinas utilizadas pelo Radix Sort reversível

Complexidade de espaço. O mecanismo de fixação faz com que cada elemento i seja movido para as listas um número total de vezes igual a $s_i + 1$: cada dígito do elemento causa um movimento; no entanto, um movimento extra e final é necessário para atribuir o elemento à lista 0 e fixá-lo. Elementos com o maior número de dígitos não chegam a ser fixados. Um elemento produz a mesma quantidade de informação sobre as listas às quais foi atribuído, ou seja, inserções em pilha, do que o espaço necessário para sua representação, que é $s_i + 1$. Consequentemente, a porção da pilha utilizada tem dimensão $O(n^*)$. Outras estruturas utilizadas têm tamanho constante ou são menores que n^* ($d < n^*$, $n < n^*$). O espaço utilizado, portanto, é $O(n^*)$.

Complexidade de tempo. A contagem de elementos, a determinação dos pontos de inserção das listas e a transformação de V_{in} em $P[1]$ executam em tempo da ordem de $O(n^*)$. A movimentação total de ponteiros entre $P[p_1]$ e $P[p_2]$ e a construção de V_{out} executam em tempo da ordem de $O(n^*)$, que é a complexidade do Algoritmo 10.

Complexidade de energia. O consumo de energia do algoritmo é zero.

8 Conclusão

Muito embora já exista literatura farta sobre o tema, a computação reversível permanece relativamente pouco difundida na comunidade científica brasileira. Movido por este desconhecimento e pelo protagonismo atual da questão energética, o presente trabalho pretendeu realizar uma introdução suficientemente abrangente do campo através da apresentação: a) de um panorama histórico da computação reversível; b) de suas bases física e lógica; c) da mecânica da reversibilidade das estruturas de controle de fluxo na máquina-modelo Pendulum; d) das definições e condições formais relativas à reversibilidade lógica; e e) de uma discussão sobre os limites práticos da computação reversível; e e) da análise de três soluções reversíveis para problemas clássicos.

Sobre a crise e a oportunidade. As questões que o trabalho sublinha têm natureza estrutural. O limite de Landauer, que se contrapõe à continuidade do aumento de eficiência dos *chips*, é decorrente tanto da física quanto da lógica irreversível tradicional e independe da implementação utilizada. Em um cenário de expansão acelerada de dispositivos inteligentes e de crescente demanda computacional por sistemas de IA, a dimensão energética alcança o primeiro plano: estudos recentes indicam que determinados *data centers* em construção no Brasil poderão consumir energia equivalente à de cidades de grande porte¹, e que a energia hoje é o grande gargalo para a expansão da IA nos EUA².

Momentos críticos geralmente cobram mudanças de paradigma. Embora o Brasil não ocupe uma posição importante na cadeia global de fabricação de semicondutores de ponta, uma eventual transição para arquiteturas reversíveis exigiria reformulações conceituais profundas. Nesse cenário, quem se antecipa colhe melhores frutos, do que depende um aumento do investimento em pesquisa.

Sobre os conceitos e as ferramentas. Do ponto de vista conceitual, este trabalho

¹<https://climainfo.org.br/2025/10/27/data-center-do-tiktok-no-ceara-consumira-mesma-energia-de-cidade-de-2-milhoes-de-habitantes/>.

²<https://www.csis.org/analysis/electricity-supply-bottleneck-us-ai-dominance>.

primeiramente apresentou a leitura consensual de que a irreversibilidade lógica leva inexoravelmente à dissipação, mas discutiu-se também, com Lairez principalmente, que esse destino não está selado, que a energia pode ser reaproveitada – porém fora do subsistema informacional.

Se, por um lado, a inversão de programas fornece regras aproveitáveis pela computação reversível, por outro, fica evidenciado que a reversibilidade não se reduz a mera execução temporal invertida. E aqui se localiza, ao nosso ver, o ponto mais importante do texto e talvez da disciplina: a tensão ou o contraste entre a reversibilidade total, representada pela arquitetura simulada na máquina Pendulum, a reversão ocorrendo por simetria estrutural, e o que seria um modelo mais realista, no qual a garantia da bijetividade é dada pela semântica ao ser adotado um modelo cujo contador de programa avança monotonicamente e a fase reversa é explicitamente construída.

Sobre os algoritmos. As versões reversíveis de algoritmos clássicos explicitaram a estratégia de bijetivização adotada em cada caso e analisaram as respectivas complexidades de tempo e de espaço obtidas.

O caso do LCS mostrou que o preenchimento da matriz de programação dinâmica não realizou sobrescrições destrutivas de modo que a versão reversível pode manter a mesma ordem de complexidade espacial da versão tradicional, sugerindo que nos casos nos quais cada entrada da matriz é preenchida uma única vez e jamais sobrescrita, o mapeamento global pode preservar informação suficiente para caracterizar a bijetividade.

No Floyd–Warshall, entretanto, a reversibilidade sem recomputações esbarra na estrutura particular do grafo de dependências do algoritmo, cujo rígido entre camadas sucessivas da computação ao longo do índice k exige que todas transformações sejam armazenadas. No entanto, utilizando *checkpointing* simples foi possível reduzir a complexidade de espaço de $\Theta(n^3)$ para $\Theta(n^{2.5})$, ao preço de um aumento apenas constante no tempo (em relação ao $\Theta(n^3)$ obtido). Tal resultado parece constituir um *trade-off* assintoticamente ótimo entre espaço e tempo para a versão reversível do algoritmo que utilize *checkpointing* simples. Destacamos que esse resultado baixou o espaço utilizado em relação ao apresentado por Demaine et al. [12] ($O(n^3)$), bem como obteve tempo melhor do que a versão reversível na qual os autores utilizaram o método da multiplicação de matrizes $(\min, +)$ para o problema dos caminhos mínimos (*All Pairs Shortest Path* – APSP) ($O(n^3 \log n)$). O acréscimo de espaço em relação ao melhor algoritmo Floyd–Warshall não reversível foi sublinear, $\sqrt{n}$.

A análise do Radix Sort reversível foi levada a cabo em termos do tamanho efetivo da entrada n^* , medido em dígitos. Nesse caso, o custo temporal justo em n^* se deve ao fato de que um elemento deixa de ser um custo após uma quantidade de iterações igual à sua quantidade de dígitos + 1. A não bijetividade característica dos procedimentos de ordenação cobra esse mesmo preço da ordem de n^* para armazenar a fila de destino do elemento em cada uma das iterações das quais participa.

Em conjunto, os resultados parecem indicar que não haja uma regra universal de reversibilização aplicável indistintamente a qualquer algoritmo. O único princípio verdadeiramente geral é a necessidade de restaurar a bijetividade global da transformação operada, através da detecção e resolução dos pontos nos quais o algoritmo original colapsa estados. Para além, a reversibilização é mais alfaia-taria do que produto de prateleira; depende da estrutura do algoritmo, da forma de representação e da dependência interna dos dados.

Sobre limitações e direções futuras. Decidimos por fazer uso exclusivo da forma estática de alocação de memória. O gerenciamento dinâmico, embora forneça soluções mais elegantes aos algoritmos, introduz problemas suficientes de modo que é necessário um estudo mais específico que aponte vantagens na sua utilização neste cenário em termos de complexidade de tempo e de espaço. Para sermos breves, a necessidade de que a liberação de memória seja uma operação reversível nega, a princípio, a própria noção de algo ser liberado³.

A abordagem utilizada no Radix Sort está intimamente ligada à decomposição dígito a dígito que lhe é característica, de modo que não é diretamente aplicável a algoritmos de ordenação baseados em comparação.

Como trabalhos futuros, propomos a investigação de técnicas para reduzir a memória auxiliar necessária para a bijetivização, através da utilização de *checkpoints em níveis hierárquicos ou multínível* e sua comparação fina com o método de Bennett [7], bem como a extensão da abordagem proposta para outras classes de algoritmos, particularmente aqueles baseados em comparações. Outra direção promissora é a exploração das compensações entre o uso de memória e a dissipação de energia em modelos computacionais semirreversíveis.

³na computação clássica, a liberação de memória pela execução de `free(p)` não devolve um ponteiro p nulo.

REFERÊNCIAS

- [1] Filippo Annovi. *A aventura de contar: da máquina de Turing à computação quântica*. Portugal: Atlântico Press, 2016. ISBN: 9789898848628.
- [2] Holger Bock Axelsen. “Clean translation of an imperative reversible programming language”. Em: *International Conference on Compiler Construction*. Springer. 2011, pp. 144–163.
- [3] Holger Bock Axelsen, Robert Glück e Tetsuo Yokoyama. “Reversible Machine Code and Its Abstract Processor Architecture”. Em: *Computer Science – Theory and Applications*. Ed. por Volker Diekert, Mikhail V. Volkov e Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 56–69.
- [4] John S. Bell. “On the Einstein Podolsky Rosen paradox”. Em: *Physics* 1.3 (1964), pp. 195–200.
- [5] Amrou Zyad Benelhaouare et al. “Thermal Side-Channel Threats in Densely Integrated Microarchitectures: A Comprehensive Review for Cyber-Physical System Security”. Em: *Micromachines* 16.10 (2025). ISSN: 2072-666X. DOI: [10.3390/mi16101152](https://doi.org/10.3390/mi16101152).
- [6] Charles H Bennett. “Logical reversibility of computation”. Em: *IBM journal of Research and Development* 17.6 (1973), pp. 525–532.
- [7] Charles H Bennett. “Time/space trade-offs for reversible computation”. Em: *SIAM Journal on Computing* 18.4 (1989), pp. 766–776.
- [8] J. S. Briggs. “Generating reversible programs”. Em: *Software: Practice and Experience* 17.7 (1987), pp. 439–453. DOI: <https://doi.org/10.1002/spe.4380170703>.
- [9] Vikraman Choudhury. *Reversible Programming Languages*. 2018.
- [10] Thomas H. Cormen et al. *Introduction to Algorithms*. 4ª ed. Cambridge, MA: MIT Press, 2022. ISBN: 9780262046305.
- [11] Joseph D’Souza. *Smart TV Statistics By Usage, Demographics and Facts*. Disponível em: <https://www.coolest-gadgets.com/smart-tv-statistics/>. Acesso em 3/3/2025. 2023.

- [12] Erik D Demaine et al. "Energy-efficient algorithms". Em: *Proceedings of the 2016 ACM conference on innovations in theoretical computer science*. 2016, pp. 321–332.
- [13] David Deutsch. "Quantum theory, the Church–Turing principle and the universal quantum computer". Em: *Proceedings of the Royal Society of London A* 400.1818 (1985), pp. 97–117.
- [14] Edsger W. Dijkstra. "Program Inversion". Em: *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer New York, 1982, pp. 351–354. ISBN: 978-1-4612-5695-3. DOI: [10.1007/978-1-4612-5695-3_63](https://doi.org/10.1007/978-1-4612-5695-3_63). URL: https://doi.org/10.1007/978-1-4612-5695-3_63.
- [15] Albert Einstein e Max Born. *The Born-Einstein Letters: Correspondence between Albert Einstein and Max and Hedwig Born from 1916 to 1955*. Tradução de Irene Born. Publicado originalmente em alemão em 1969. New York: Walker, 1971.
- [16] O Estadão. *Número de celulares conectados a internet deve subir de 4,7 bi para 5,5 bi até 2023, prevê GSMA*. Disponível em: <https://epocanegocios.globo.com/tecnologia/noticia/2024/02/numero-de-celulares-conectados-ainternet-deve-subir-de-47-bi-para-55-bi-ate-2030-preve-gsma.ghtml>. Acesso em 21/04/2024. 2024.
- [17] Richard P. Feynman. "Simulating physics with computers". Em: *International Journal of Theoretical Physics* 21.6-7 (1982), pp. 467–488.
- [18] Robert Floyd. "Algorithm 97: shortest path". Em: *Communications of the ACM* 5.6 (1962), pp. 345–345.
- [19] Robert W Floyd. "Assigning meanings to programs". Em: *Program Verification: Fundamental Issues in Computer Science*. Springer, 1993, pp. 65–81.
- [20] Michael Frank. "Approaching the physical limits of computing". Em: *35th International Symposium on Multiple-Valued Logic (ISMVL'05)*. IEEE. 2005, pp. 168–185.
- [21] Michael P Frank. *Back to the future: the case for reversible logic*. 2017. URL: <https://arxiv.org/abs/1803.02789>.
- [22] Michael P Frank. "Foundations of generalized reversible computing". Em: *International Conference on Reversible Computation*. Springer. 2017, pp. 19–34.
- [23] Michael P Frank e Thomas F Knight Jr. "Reversibility for efficient computing". Tese de dout. Massachusetts Institute of Technology, Dept. of Electrical Engineering e Computer Science, 1999.

- [24] Edward Fredkin e Tommaso Toffoli. “Conservative logic”. Em: *International Journal of theoretical physics* 21.3 (1982), pp. 219–253.
- [25] Dina Genkina. *Reversible Computing Escapes the Lab in 2025*. Disponível em: <https://spectrum.ieee.org/reversible-computing>. Acesso em 22/5/2025. 2025.
- [26] Robert Glück e Tetsuo Yokoyama. “Reversible computing from a programming language perspective”. Em: *Theoretical Computer Science* 953 (2023).
- [27] Diana Goovaerts. *CoolIT ramps its manufacturing capacity by 25x*. Disponível em: <https://www.fierce-network.com/data-center/coolit-ramps-its-manufacturing-capacity-25x>. Acesso em 11/4/2025. 2024.
- [28] Cindy Gordon. *ChatGPT And Generative AI Innovations Are Creating Sustainability Havoc*. Disponível em: <https://www.forbes.com/sites/cindygordon/2024/03/12/chatgpt-and-generative-ai-innovations-are-creating-sustainability-havoc/>. Acesso em 12/3/2025. 2024.
- [29] David Gries. “Inverting programs”. Em: *The Science of Programming* (1981), pp. 265–274.
- [30] Paul E Gronowski et al. “High-performance microprocessor design”. Em: *IEEE Journal of Solid-State Circuits* 33.5 (2002), pp. 676–686.
- [31] Godfrey Harold Hardy, John Edensor Littlewood e George Pólya. *Inequalities*. Cambridge: Cambridge University Press, 1934. ISBN: 978-0521358804.
- [32] Tue Haulund. “Design and implementation of a reversible object-oriented programming language”. Diss. de mest. Faculty of Science - University of Copenhagen, 2017. URL: <https://arxiv.org/abs/1707.07845>.
- [33] C. A. R. Hoare. “An axiomatic basis for computer programming”. Em: *Commun. ACM* 12.10 (out. de 1969), pp. 576–580. ISSN: 0001-0782. DOI: [10.1145/363235.363259](https://doi.org/10.1145/363235.363259). URL: <https://doi.org/10.1145/363235.363259>.
- [34] Robin Kaarsgaard. “The Logic of Reversible Computing: Theory and Practice”. English. Tese de dout. Department of Computer Science, Faculty of Science, University of Copenhagen, 2017.
- [35] Jeremy Kahn. *Startup working on ‘reversible computing’ chip for AI says initial tests show a 50% energy savings*. Disponível em: <https://fortune.com/2025/05/20/uk-startup-vaire-reversible-computing-chip-gpu-alternative-energy-savings-ai/?ref=biztoc.com>. Acesso em 21/7/2025. 2025.

- [36] Phillip Kaye, Raymond Laflamme e Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.
- [37] Brian W. Kernighan e Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [38] Didier Lairez. “Thermodynamical versus logical irreversibility: A concrete objection to Landauer’s principle”. Em: *Entropy* 25.8 (2023), p. 1155.
- [39] R. Landauer. “Irreversibility and Heat Generation in the Computing Process”. Em: *IBM Journal of Research and Development* 5.3 (1961), pp. 183–191. doi: [10.1147/rd.53.0183](https://doi.org/10.1147/rd.53.0183).
- [40] Yves Lecerf. “Machines de Turing réversibles”. Em: *Comptes Rendus Hebdomadaires des Séances de l’Académie des Sciences* 257 (1963), pp. 2597–2600.
- [41] George B Leeman Jr. “A formal approach to undo operations in programming languages”. Em: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.1 (1986), pp. 50–87.
- [42] Martin Lukac et al. “Building a Completely Reversible Computer”. Em: *arXiv preprint arXiv:1702.08715* (2017).
- [43] Christopher Lutz e Howard Derby. “Janus: a time-reversible language”. Em: *Letter to R. Landauer* 2 (1986). Disponível em <https://tetsuo.jp/ref/janus.pdf>, acessado em 07/01/2025.
- [44] N. David Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007.
- [45] Steven Gonzalez Monserrate. *The Staggering Ecological Impacts of Computation and the Cloud*. Disponível em: <https://thereader.mitpress.mit.edu/the-staggering-ecological-impacts-of-computation-and-the-cloud/>. Acesso em 11/4/2025. 2025.
- [46] Gordon E Moore. “Cramming more components onto integrated circuits”. Em: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.
- [47] Marcos Moura. “Entropia estatística e o ensino da segunda lei da termodinâmica (Apêndice A: Entropia e a segunda lei da termodinâmica)”. Tese de dout. UFRJ, 2023. URL: https://www.if.ufrj.br/~carlos/teses/tese_Marcos_Moura.pdf.
- [48] Johann von Neumann e Arthur W Burks. *Theory of self-reproducing automata*. University of Illinois Press Urbana, 1966.

- [49] Michael A. Nielsen e Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.
- [50] Karmela Padavic-Callaghan. *How to fix computing's AI energy problem: run everything backwards*. Disponível em: <https://www.newscientist.com/article/mg26435231-300-how-to-fix-computings-ai-energy-problem-run-everything-backwards/>.. 2024.
- [51] Kalyan S Perumalla e Richard M Fujimoto. *Source-code transformations for efficient reversibility*. Rel. técn. Georgia Institute of Technology, 1999.
- [52] Kalyan S. Perumalla. *Introduction to reversible computing*. Chapman e Hall/CRC Press, 2013.
- [53] John Preskill. "Lecture notes for physics 229: Quantum information and computation". Em: *California Institute of Technology* (1998).
- [54] Jan M Rabaey, Anantha Chandrakasan e Borivoje Nikolic. *Digital integrated circuits*. Vol. 2. Prentice hall Englewood Cliffs, 2002.
- [55] Susan Rakov. *Fact file: Computing is using more energy than ever*. Disponível em: <https://frontiergroup.org/resources/fact-file-computing-is-using-more-energy-than-ever/>. Acesso em 21/4/2025. 2023.
- [56] Ravi Sethi. "Complete Register Allocation Problems". Em: *SIAM Journal on Computing* 4.3 (1975), pp. 226–248. doi: [10.1137/0204020](https://doi.org/10.1137/0204020).
- [57] Claude E Shannon. "A mathematical theory of communication". Em: *The Bell system technical journal* 27.3 (1948), pp. 379–423.
- [58] Paul Shepard. 'Intrachip' Micro-Cooling System for High-Performance Electronics. Disponível em: <https://eepower.com/news/intrachip-micro-cooling-system-for-high-performance-electronics/>. Acesso em 26/5/2025. 2017.
- [59] Peter W Shor. "Algorithms for quantum computation: discrete logarithms and factoring". Em: *Proceedings 35th annual symposium on foundations of computer science*. IEEE. 1994, pp. 124–134.
- [60] Michael Kirkedal Thomsen, Holger Bock Axelsen e Robert Glück. "A Reversible Processor Architecture and Its Reversible Logic Design". Em: *Reversible Computation*. Ed. por Alexis De Vos e Robert Wille. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 30–42. ISBN: 978-3-642-29517-1.
- [61] Tommaso Toffoli. "Reversible computing". Em: *International colloquium on automata, languages, and programming*. Springer. 1980, pp. 632–644.

- [62] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. Em: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: <https://doi.org/10.1112/plms/s2-42.1.230>.
- [63] A. M. Turing. “Systems of logic based on ordinals”. Tese de dout. Princeton University, 1938. URL: <https://turing.academicwebsite.com/publications/19-systems-of-logic-based-on-ordinals>.
- [64] Nirvan Tyagi, Jayson Lynch e Erik D Demaine. “Toward an energy efficient language and compiler for (partially) reversible algorithms”. Em: *International Conference on Reversible Computation*. Springer. 2016, pp. 121–136.
- [65] Vaire. *A software-first paradigm for computing architectures*. Rel. técn. Vaire Computing Inc., 2025. URL: <https://vaire.co/whitepapers>.
- [66] Vaire. *The Technology of Near-Zero Energy Computing*. Rel. técn. Vaire Computing Inc., 2025. URL: <https://vaire.co/whitepapers>.
- [67] Yuri Vasconcelos. *Os impactos ambientais da computação*. Disponível em: <https://revistapesquisa.fapesp.br/os-impactos-ambientais-da-computacao/>. Acesso em 22/4/2025. 2025.
- [68] Carlin Vieri et al. “A fully reversible asymptotically zero energy microprocessor”. Em: *Power Driven Microarchitecture Workshop*. University of Michigan. 1998, pp. 138–142.
- [69] Carlin James Vieri. “Pendulum—a reversible computer architecture”. Diss. de mest. Massachusetts Institute of Technology, 1995.
- [70] Carlin James Vieri e Thomas F. Knight. “Reversible computer engineering and architecture”. AAI0800892. Tese de dout. USA: Massachusetts Institute of Technology, 1999.
- [71] Tetsuo Yokoyama, Holger Bock Axelsen e Robert Glück. “Principles of a reversible programming language”. Em: *Proceedings of the 5th Conference on Computing Frontiers*. 2008, pp. 43–54.
- [72] Tetsuo Yokoyama e Robert Glück. “A reversible programming language and its invertible self-interpreter”. Em: *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 2007, pp. 144–153.
- [73] Saed G Younis. “Asymptotically zero energy computing using split-level charge recovery logic”. Tese de dout. Massachusetts Institute of Technology, 1994.

APÊNDICE A - Elementos de Computação Quântica

A.1 Motivação

Na seção 3.1 citamos uma passagem de Michael P. Frank [23] na qual afirma que a reversibilidade é uma propriedade das leis mais fundamentais da física, presente inclusive na física quântica. Isso quer dizer que, se a evolução dos estados quânticos é conhecida, é possível "girar o relógio ao contrário" e obter os estados passados [49, 53].

A computação quântica é um modelo de computação baseado nos princípios da mecânica quântica, no qual a evolução dos estados do sistema é reversível – embora a medição introduza irreversibilidade [49]. Essa característica conecta naturalmente a computação quântica com a computação reversível clássica, na qual cada transformação é bijetiva [6, 49].

Neste apêndice apresentamos apenas o modelo abstrato de computação quântica, concentrando-nos nas definições principais e nos efeitos da aplicação de algumas portas quânticas sobre a unidade fundamental desta disciplina.

A.2 O *qubit* e os estados básicos

No computador clássico, o estado da memória é exatamente aquilo que é obtido com sua medição: há aí um determinismo. No computador quântico, a natureza probabilística de medição introduz uma dinâmica distinta. O estado da unidade básica do computador quântico, o *qubit*, não corresponde diretamente ao resultado da medição, mas a um conjunto de amplitudes complexas que determinam probabilisticamente os resultados possíveis. O *qubit* é fisicamente composto por um pedaço de matéria ou radiação cujo comportamento é descrito pelas leis da mecânica quântica.

Um *qubit* é representado por um vetor unitário em um espaço de Hilbert complexo bidimensional [49, 44, 53]. Seu estado é definido como:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \text{s.a.: } |\alpha|^2 + |\beta|^2 = 1, \quad \alpha, \beta \in \mathbb{C}$$

$|\alpha|^2$ e $|\beta|^2$ representam, respectivamente, as probabilidades de uma medição do *qubit* obter o resultado 0 ou 1 [49, 44]. Esses dois resultados possíveis formam o que chamamos de *estados básicos* ou *base computacional* de um *qubit*, que convencionalmente representamos como $|0\rangle$ e $|1\rangle$. Antes da medição, o *qubit* permanece em uma *superposição linear* desses dois estados, na qual seus coeficientes não são diretamente observáveis, mas apenas as probabilidades associadas a eles [49, 36]. Trata-se de um estado único e coeso, que, ao ser medido, é colapsado probabilisticamente em um dos estados da base. Embora a medição seja inerentemente probabilística, o controle preciso das amplitudes α e β permite que algoritmos quânticos aumentem a probabilidade de resultados desejados [49, 59], o que constitui uma das principais fontes de vantagem deste modelo de computação. O *qubit* é melhor visualizado através da esfera de Bloch [49, 44], ilustrada na Figura 31, juntamente com as informações intrínsecas.

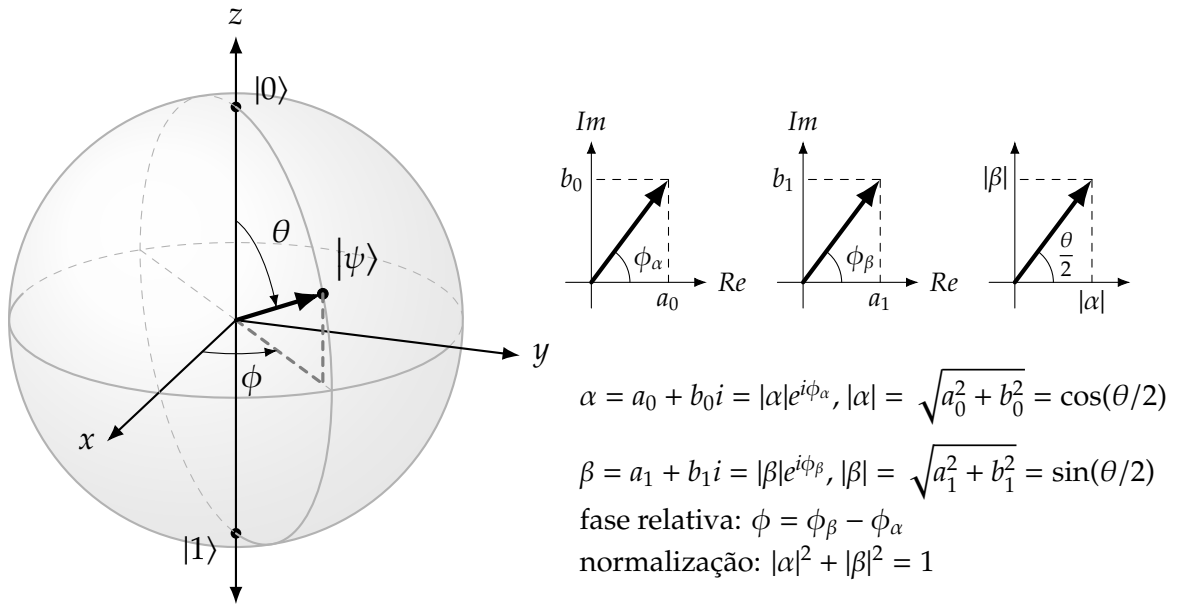


Figura 31: Representação do *qubit* através da esfera de Bloch. Note-se que a representação do vetor em um espaço no qual os estados $|0\rangle$ e $|1\rangle$ são ortogonais faz com que o ângulo obtido seja a metade daquele que na esfera corresponde a θ .

A.3 Números complexos

Um número complexo α pode ser escrito de duas formas:

- a cartesiana, $a + bi$, na qual há a parte real (a) e a imaginária (b); e
- a polar, p.ex., $|\alpha|e^{i\phi}$, na qual participam o módulo do vetor que representa o complexo e a fase, esta correspondendo ao ângulo ϕ de inclinação do vetor em relação ao eixo Real (Re)¹.

Multiplicar um complexo por $e^{i\phi}$ significa rotacionar o vetor pelo ângulo ϕ [49], o que implica no reajuste das coordenadas a e b para manutenção do mesmo módulo².

Exemplo:

$$\begin{aligned}\alpha &= |\alpha|e^{i\phi} = 2e^{i0} = 2 + 0i = 2, & |\alpha| &= 2, \phi = 0^\circ (a = 2, b = 0) \\ |\alpha|e^{i\pi/4} &= 2(\cos(\pi/4) + i \sin(\pi/4)) = \sqrt{2} + \sqrt{2}i, & |\alpha| &= 2, \phi = 45^\circ (a = b = \sqrt{2})\end{aligned}$$

De forma análoga, multiplicar um estado quântico por um fator $e^{i\phi}$ aplica uma fase global, não alterando a fase relativa entre suas amplitudes [49, 53] – isto é, a diferença $\phi = \phi_\beta - \phi_\alpha$ permanece inalterada, pois ambas são rotacionadas igualmente.

Operações de fase relativa alteram a longitude (ϕ) na esfera de Bloch, preservando a latitude (θ), pois não modificam os módulos $|\alpha|$ e $|\beta|$. Dessa forma, não alteram diretamente as probabilidades de medição, mas influenciam os efeitos de *interferência* quando combinadas com outras operações que misturam amplitudes. Interferência é o fenômeno pelo qual amplitudes complexas se somam, podendo reforçar ou cancelar probabilidades em função de suas fases relativas [49, 17] – ver A.4.1, porta H.

Essas operações podem ser interpretadas como uma etapa de preparação do estado, cujo efeito se torna observável apenas quando seguidas por transformações que alteram as amplitudes, modificando as probabilidades $P(0)$ e $P(1)$. Estas últimas executam transformações do tipo $\alpha \leftarrow f_1(\alpha, \beta)$, $\beta \leftarrow f_2(\alpha, \beta)$.

¹Utilizando Pitágoras, $|\alpha| = \sqrt{a^2 + b^2}$.

²Lembrar-se de que: $e^{i\phi} = \cos \phi + i \sin \phi$. Valores notáveis: $e^{i0} = 1$; $e^{i\pi/2} = i$ (rotaciona 90°); $e^{i\pi} = -1$ (180°); $e^{i3\pi/2} = -i$ (270°); $e^{i2\pi} = 1$ (rotação completa).

A.4 Portas quânticas

Elencamos abaixo algumas das infinitas portas quânticas possíveis [49, 44]³.

A.4.1 Portas de *qubit* único (*Single-qubit gates*)

- **I (Identidade)**. Retorna o próprio *qubit*.

$$I|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

$$P(0) = |\alpha|^2, \quad P(1) = |\beta|^2$$

- **X (Pauli-X)**. Equivale ao NOT clássico, rotacionando o *qubit* em 180° no eixo x da esfera de Bloch⁴. Com isso, inverte as amplitudes α e β e as fases ϕ_α e ϕ_β .

$$X|\psi\rangle = \beta|0\rangle + \alpha|1\rangle, \quad P(0) = |\beta|^2, P(1) = |\alpha|^2$$

- **Z (Pauli-Z)**. Rotaciona o *qubit* em 180° no eixo z . A latitude, e consequentemente, as probabilidades não são alteradas. Mas a fase de β é invertida⁵:

$$Z|\psi\rangle = \alpha|0\rangle - \beta|1\rangle, \quad \beta e^{i\pi} = -\beta$$

$$P(0) = |\alpha|^2, \quad P(1) = |-\beta|^2 = |\beta|^2$$

- **Y (Pauli-Y)**. Rotaciona o *qubit* em 180° no eixo y , invertendo as amplitudes e introduzindo um fator de fase imaginário. Pode ser escrita como $Y = iXZ$, o que expressa uma relação algébrica entre os operadores de Pauli, e não uma decomposição em rotações sequenciais.

$$\alpha|0\rangle + \beta|1\rangle \xrightarrow{Z} \alpha|0\rangle - \beta|1\rangle \xrightarrow{Y} -\beta|0\rangle + \alpha|1\rangle \xrightarrow{i} Y|\psi\rangle = -i\beta|0\rangle + i\alpha|1\rangle$$

$$P(0) = |-i\beta|^2 = |\beta|^2, \quad P(1) = |i\alpha|^2 = |\alpha|^2$$

- **H (Hadamard)**. Combinação linear dos operadores X e Z , o que corresponde, geometricamente, a uma rotação de π em torno de um eixo no plano XZ , inclinado

³Dado que parâmetros como ângulo $\theta \in \mathbb{R}$ podem ser utilizados em rotações, o número de portas possíveis tem cardinalidade $2^{\aleph_0}$, que é a do contínuo – $\aleph_0$ sendo a cardinalidade de $\mathbb{N}$.

⁴Para melhor visualização: mexer na latitude altera as amplitudes; no limite (180°), elas se invertem.

⁵Como a fase relativa $\phi = \phi_\beta - \phi_\alpha$, qualquer alteração desejada nesse valor opera somente sobre ϕ_β , ϕ_α sendo tratado como um referencial fixo.

a 45° entre esses eixos. Esse movimento troca os papéis de amplitude e fase relativa, permitindo que diferenças de fase sejam convertidas em diferenças de probabilidade observáveis [49, 44].

$$\mathbf{H}|\psi\rangle = \frac{(\alpha + \beta)}{\sqrt{2}}|0\rangle + \frac{(\alpha - \beta)}{\sqrt{2}}|1\rangle, \quad P(0) = \frac{|\alpha + \beta|^2}{2}, P(1) = \frac{|\alpha - \beta|^2}{2}$$

As probabilidades podem ser escritas na forma alternativa:

$$P(0) = \frac{1}{2} + |\alpha||\beta| \cos \phi, \quad P(1) = \frac{1}{2} - |\alpha||\beta| \cos \phi \quad (\text{s.a. : } |\alpha|^2 + |\beta|^2 = 1)$$

Dois tipos de interferência podem ser esperadas: a) construtiva: as fases alinhadas se somam, aumentando a probabilidade de se observar o estado; e b) destrutiva: as fases opostas fazem as amplitudes diminuírem ou se cancelarem, reduzindo a probabilidade de medição do estado [49, 17].

Como é a sua própria inversa ($\mathbf{H}^2 = \mathbf{I}$), a porta é utilizada tanto para criar superposição a partir de estados inicialmente puros ($\mathbf{H}|0\rangle \rightarrow |+\rangle$, $\mathbf{H}|1\rangle \rightarrow |-\rangle$)⁶, quanto para gerar estados puros a partir de superposição ($\mathbf{H}|+\rangle \rightarrow |0\rangle$, $\mathbf{H}|-\rangle \rightarrow |1\rangle$) – ver seção A.6.

- **S.** Rotaciona o eixo z em 90° , aplicando uma fase relativa de $\pi/2$ ao estado $|1\rangle$. Ao contrário das portas de Pauli, a porta S não é sua própria inversa, equivalendo a meia porta Z ($S^2 = Z$). Sua rotação inclui a fase imaginária i .

$$\mathbf{S}|\psi\rangle = \alpha|0\rangle + i\beta|1\rangle$$

$$P(0) = |\alpha|^2, \quad P(1) = |i\beta|^2 = |\beta|^2$$

- **T.** Rotaciona o eixo z em 45° , aplicando uma fase relativa de $\pi/4$. Equivale a meia porta S ($\mathbf{T}^2 = \mathbf{S}$).

$$\mathbf{T}|\psi\rangle = \alpha|0\rangle + e^{i\pi/4}\beta|1\rangle$$

$$P(0) = |\alpha|^2, \quad P(1) = |\beta|^2$$

⁶Os estados $|+\rangle$ e $|-\rangle$ são estados de superposição equilibrada, nos quais $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, e $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. $|+\rangle$ é o resultado da aplicação da *Hadamard* ao estado inicial $|0\rangle$: $\mathbf{H}|0\rangle = |+\rangle$; da mesma forma, $\mathbf{H}|1\rangle = |-\rangle$.

A.4.2 Portas de *qubit* múltiplo (*Multiple-qubit gates*)

Neste ponto, a análise transita das rotações de estado individual para as operações de controle lógico. Com este conjunto de portas, a computação quântica torna-se capaz de emular o comportamento da lógica booleana clássica de forma reversível [6, 61].

Um estado geral de n *qubits* é um vetor em um espaço de Hilbert de dimensão 2^n [49, 36]:

$$|\Psi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x |x\rangle, \quad \sum_{x \in \{0,1\}^n} |\alpha_x|^2 = 1$$

que pode, no caso de estados separáveis, ser fatorado como produto tensorial de estados individuais [49]⁷:

$$|\Psi\rangle = \bigotimes_{i=1}^n |\psi_i\rangle$$

De modo que o produto de três *qubits* pode ser escrito formalmente: $|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle$; em justaposição: $|\psi_1\rangle|\psi_2\rangle|\psi_3\rangle$; ou na forma compacta: $|\psi_1\psi_2\psi_3\rangle$. A tabela verdade de uma porta é obtida a partir da aplicação da regra distributiva sobre os termos da escrita do produto tensorial juntamente com a operação da porta sobre os *qubits* alvo, lembrando que os estados individuais mais utilizados são:

$$|0\rangle = |0\rangle + |1\rangle, \quad |1\rangle = |0\rangle + |1\rangle, \quad |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

- **CNOT**. Inverte o estado do alvo t (aplica porta **X** se, e somente se, o controle c for $|1\rangle$).

$$\text{CNOT}|c, t\rangle = |c, c \oplus t\rangle$$

- Se $c \in \{|0\rangle, |1\rangle\}$, o estado resultante para $t \in \{|0\rangle, |1\rangle\}$.

⁷Estados que não admitem tal fatoração são chamados de estados emaranhados, cuja construção e propriedades serão exploradas adiante.

- Se $c \in \{|+\rangle, |-\rangle\}$ e $t \in \{|0\rangle, |1\rangle\}$, é produzido o fenômeno conhecido como *emaranhamento quântico* [49, 53]: a partir da aplicação da transformação e por um tempo indeterminado enquanto o sistema estiver isolado do ambiente (mantiver *coerência*), a medição de um *qubit* fixa instantaneamente o resultado do outro de forma correlacionada, mas sem transmissão de informação [4, 49]⁸.

Exemplo: Se a entrada for $|+,0\rangle$ e a medição resultar $c = 1$, o colapso de c para um estado clássico faz com que t também colapse instantaneamente para $t = 1$ – porque t depende de c , não o contrário. Isso valerá mesmo que c e t sejam separados espacialmente por uma grande distância⁹.

A Tabela 6 resume os efeitos descritos acima.

Entrada	Saída	Efeito Lógico / Quântico
$ +,0\rangle$	$\frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$	Estado de Bell $ \Phi^+\rangle$
$ +,1\rangle$	$\frac{1}{\sqrt{2}}(01\rangle + 10\rangle)$	Estado de Bell $ \Psi^+\rangle$
$ -,0\rangle$	$\frac{1}{\sqrt{2}}(00\rangle - 11\rangle)$	Estado de Bell $ \Phi^-\rangle$
$ -,1\rangle$	$\frac{1}{\sqrt{2}}(01\rangle - 10\rangle)$	Estado de Bell $ \Psi^-\rangle$

Tabela 6: Resumo operacional da porta CNOT para o caso c em superposição e o alvo t não, evidenciando o emaranhamento quântico (*entanglement*).

Ver seção A.5 sobre cálculos utilizando matrizes unitárias e produtos tensoriais.

- Se $c \in \{|+\rangle, |-\rangle\}$ e $t \in \{|+\rangle, |-\rangle\}$, o resultado é o mesmo da entrada a menos que a última seja $|1,-\rangle$, quando então uma fase relativa é transferida para c (*phase kickback*):

$$|1,-\rangle = |1\rangle \otimes \left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \right] = \frac{1}{\sqrt{2}}(|10\rangle - |11\rangle)$$

$$\text{CNOT } |1,-\rangle = \frac{1}{\sqrt{2}}(|1,1 \oplus 0\rangle - |1,1 \oplus 1\rangle) = \frac{1}{\sqrt{2}}(|11\rangle - |10\rangle) = -|1,-\rangle$$

- Se $c, t \in \{|+\rangle, |-\rangle\}$, o resultado é o mesmo da entrada a menos que a última seja $|+,-\rangle$, quando então uma fase relativa é transferida para c ($|+,-\rangle$):

⁸De uma forma geral, o emaranhamento em uma porta controlada acontece quando o controle está em superposição e o estado do alvo é tal que ele muda com a aplicação da operação da porta. A sua ocorrência faz com que a escrita do resultado altere a distribuição de pesos entre os componentes ou faça desaparecer alguns termos – ver Tabela 6.

⁹A mecânica quântica não apresenta um limite teórico para isso que Albert Einstein chamou de *ação fantasmagórica* (*spukhafte Fernwirkung*) [15, 4], desde que a *coerência* seja garantida durante todo o trajeto: o *qubit* deve continuar isolado do ambiente, sob pena de colapsar.

$$\begin{aligned}
|+, -\rangle &= \left[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \right] \otimes \left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \right] = \frac{1}{2} [|0\rangle(|0\rangle - |1\rangle) + |1\rangle(|0\rangle - |1\rangle)] = \\
&\quad \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \\
\text{CNOT}|+, -\rangle &= \frac{1}{2}(|0, 0 \oplus 0\rangle - |0, 0 \oplus 1\rangle + |1, 1 \oplus 0\rangle - |1, 1 \oplus 1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = \\
&\quad \frac{1}{2} [|0\rangle(|0\rangle - |1\rangle) - |1\rangle(|0\rangle - |1\rangle)] = \left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \right] \otimes \left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \right] = |-, -\rangle
\end{aligned}$$

- **Toffoli (CCNOT ou C^2X).** É a porta CNOT ampliada para 3 *qubits*, que inverte o alvo t caso os estados dos controles c_1 e c_2 sejam $|1\rangle$.

$$C^2X|c_1, c_2, t\rangle = |c_1, c_2, t \oplus (c_1 \cdot c_2)\rangle$$

A Tabela 7 resume alguns casos específicos os efeitos descritos abaixo.

- Quando $t, c_1, c_2 \in \{|0\rangle, |1\rangle\}$, a porta funciona como a sua homônima clássica reversível.
- Quando $c_1 = c_2 = |+\rangle$ e o alvo em estado clássico é produzido emaranhamento, de modo que a medição do estado do alvo determina os estados dos controles. Exemplo:

$$\begin{aligned}
|+, +, 0\rangle &= \left[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \right]^{\otimes 2} \otimes |0\rangle = \frac{1}{2}(|000\rangle + |010\rangle + |100\rangle + |110\rangle) \\
C^2X|+, +, 0\rangle &= \frac{1}{2}(|00, 0 \oplus 0 \cdot 0\rangle + |01, 0 \oplus 0 \cdot 1\rangle + |10, 0 \oplus 1 \cdot 0\rangle + |11, 1 \oplus 1 \cdot 1\rangle) = \\
&\quad \frac{1}{2}(|000\rangle + |010\rangle + |100\rangle + |111\rangle)
\end{aligned}$$

Nota-se que o emaranhamento faz com que o resultado $t = 1$ force o colapso $c_1 = c_2 = 1$. Se $t = 0$, o estado colapsa para uma superposição dos outros 3 termos¹⁰. Similarmente, quando a entrada é $|+, +, 1\rangle$, $t = 0$ força o mesmo colapso ($c_1 = c_2 = 1$) – ver seção A.5 sobre produto tensorial e matrizes unitárias.

- Se $c_1, c_2 \in \{|0\rangle, |1\rangle\}$ e $t = |-\rangle$, a porta não altera o resultado, a menos que $c_1 = c_2 = 1$, quando então $C^2X|11, -\rangle = -|11, -\rangle$ ¹¹.
- Se $c_1, c_2 \in \{|+, +\rangle, |-, -\rangle\}$, a porta isola e inverte a fase da componente $|11\rangle$ e emaranha os controles no restante dos estados, com a exceção do caso

¹⁰ $\frac{1}{\sqrt{3}}(|000\rangle + |010\rangle + |100\rangle)$.

¹¹ No caso $|11, +\rangle$, como $X|+\rangle = |+\rangle$, o resultado não se altera. Embora a representação na Esfera de Bloch sugira que os estados $|+\rangle$ e $|-\rangle$ são igualmente invariantes sob a porta X , a análise dos autovalores revela uma inversão de fase crucial no estado $|-\rangle$. Essa distinção algébrica, invisível na geometria individual do *qubit*, é o que permite à porta C^2X realizar operações de lógica de fase em algoritmos complexos.

$|+, +, +\rangle$ ¹². A porta não altera o alvo se os controles estiverem em fases não alinhadas ($|+, -\rangle$ ou $|-, +\rangle$).

Entrada	Saída	Tipo de Estado	Efeito Lógico / Quântico
$ 110\rangle$	$ 111\rangle$	Produto	Porta AND reversível
$ 111\rangle$	$ 110\rangle$	Produto	NOT condicional
$ +, +, 0\rangle$	$\frac{1}{2}(000\rangle + 010\rangle + 100\rangle + 111\rangle)$	Emaranhado	Geração de correlação tripartida
$ 11, -\rangle$	$- 11, -\rangle$	Produto	Inversão da fase global
$ +, +, -\rangle$	$\frac{1}{2}(00\rangle + 01\rangle + 10\rangle - 11\rangle) \otimes -\rangle$	Emaranhado*	Marcação de fase do termo $ 11\rangle$
$ 01, +\rangle$	$ 01, +\rangle$	Produto	Invariância por controle nulo ($c_1 = 0$)
$ 11, +\rangle$	$ 1, 1, +\rangle$	Produto	$X +\rangle = +\rangle$

*Emaranhamento restrito aos qubits de controle no sistema reduzido.

Tabela 7: Mapeamento de estados notáveis na porta *Toffoli* (C^2X).

- **SWAP**. Executa uma operação de permutação sobre as informações de dois *qubits*, preservando a natureza do estado (produto ou emaranhado), independentemente de estes se encontrarem ou não em superposição.

$$\text{SWAP } |t_1, t_2\rangle = |t_2, t_1\rangle$$

- **Fredkin (CSWAP)**. Realiza SWAP caso c esteja em $|1\rangle$.

$$\text{CSWAP } |c, t_1, t_2\rangle = \begin{cases} |0, t_1, t_2\rangle, & \text{se } c = 0 \\ |1, t_2, t_1\rangle, & \text{se } c = 1 \end{cases}$$

O emaranhamento máximo ocorre com: $|+\rangle \otimes |01\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |01\rangle = \frac{1}{\sqrt{2}}(|001\rangle + |110\rangle)$: o conhecimento do estado de qualquer *qubit* força o conhecimento do estado dos outros dois. A Tabela 8 apresenta os casos notáveis.

Entrada	Saída	Estado	Efeitos Lógicos / Quânticos
$ +, 01\rangle$	$\frac{1}{\sqrt{2}}(001\rangle + 110\rangle)$	Emaranhado	Geração de correlação tripartida
$ +, +, -\rangle$	$\frac{1}{\sqrt{2}}(0, +, -\rangle + 1, -, +\rangle)$	Emaranhado	Inversão de fase dos alvos quando $c = 1$
$ 1, \psi, \psi\rangle$	$ c, \psi, \psi\rangle$	Produto	A troca de estados idênticos é invariante
$ 1, +, -\rangle$	$ 1, -, +\rangle$	Produto	Permutação pura
$ 0, \psi, \phi\rangle$	$ 0, \psi, \phi\rangle$	Produto	Invariância por controle nulo ($c = 0$)

Tabela 8: Análise de Estados Notáveis na Porta *Fredkin* (CSWAP)

¹²O emaranhamento não ocorre pois, como já visto, $X|+\rangle = |+\rangle$, então o alvo não se altera.

A.5 Matrizes de permutação unitárias e produtos tensoriais

As portas quânticas podem ser representadas por matrizes de permutação unitárias, definidas estas como as que possuem somente uma ocorrência de 1 em cada linha, os demais valores sendo iguais a 0. Pela regra da reversibilidade, cada coluna conterá apenas um valor 1 [49].

Utilizaremos a porta CNOT como exemplo. Sua matriz de permutação unitária é:

$$\begin{array}{llll} \text{CNOT } |00\rangle & |00\rangle & \leftarrow 1^{\text{a}} \text{ entrada} \rightarrow & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \\ \text{CNOT } |01\rangle & |01\rangle & \leftarrow 2^{\text{a}} \text{ entrada} \rightarrow & \\ \text{CNOT } |10\rangle & |11\rangle & \leftarrow 4^{\text{a}} \text{ entrada} \rightarrow & \\ \text{CNOT } |11\rangle & |10\rangle & \leftarrow 3^{\text{a}} \text{ entrada} \rightarrow & \end{array} = U_{\text{CNOT}}$$

Note-se que $U_{\text{CNOT}}^2 = I$, a matriz identidade. No caso em que $|+,0\rangle$, temos:

$$|+\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \Rightarrow |+\rangle \otimes |0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Então, o resultado da aplicação da operação CNOT sobre o par $|+,0\rangle$ é:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle)$$

A.6 Programação quântica

A evolução de sistemas quânticos isolados é descrita por operadores unitários U que, como vimos nas seções anteriores, satisfazem $U^\dagger U = I$ [49, 53]. Transformações unitárias são sempre reversíveis. Assim, algoritmos quânticos são implementados por sequências de portas unitárias aplicadas aos *qubits* [49, 13] devidamente inicializados em $|0\rangle$. A cada passo da execução, as fases e amplitudes são manipuladas de forma a aumentarem a probabilidade de medição do resultado final esperado.

O emaranhamento, na busca desse objetivo, permite que o estado do sistema seja descrito de forma global, não fatorável em componentes independentes. Com isso, as operações quânticas passam a atuar sobre o conjunto completo de *qubits*, manipulando

de forma conjunta suas amplitudes e fases relativas. Essa dinâmica é essencial para a aplicação da interferência, que aumenta a probabilidade em resultados de interesse. Diferentemente de um processamento independente de cada configuração, o comportamento do sistema passa a depender das correlações internas entre seus componentes. O sistema deixa, então, de evoluir como um conjunto de estados isolados e passa a se comportar como uma única entidade.

Alguns sítios virtuais que permitem a experimentação da programação quântica são:

- IBM Composer (<https://quantum.cloud.ibm.com/composer>)¹³;
- Quirk (<https://algassert.com/quirk>); e
- Quantum Flytrap (<https://quantumflytrap.com/virtual-lab/>).

A.7 Relação com a computação reversível clássica

Funções clássicas reversíveis podem ser expressas como operadores lineares que realizam permutações nos estados da base computacional. Um exemplo fundamental é o operador U_f , definido por:

$$U_f |x, y\rangle = |x, y \oplus f(x)\rangle$$

Na fórmula, a preservação do estado de entrada $|x\rangle$ e o uso da operação XOR ($\oplus$) garantem que a computação seja reversível. Do ponto de vista matemático, esses operadores são representados por matrizes de permutação – um subconjunto das matrizes unitárias [49, 36]. Logo, circuitos reversíveis clássicos podem ser interpretados como casos particulares de circuitos quânticos que operam estritamente sobre a base computacional [36].

Resumindo, toda computação reversível clássica é um subconjunto da computação quântica. Operações bijetivas clássicas correspondem a transformações unitárias restritas à base computacional (0 e 1), enquanto a computação quântica permite ainda manipular superposições de estados sem perder reversibilidade [6, 49, 36].

¹³Video com um exemplo de uso: <https://www.youtube.com/watch?v=v6fZgAF-Le8&t=2s>.