



ALGORITMOS DE TRANSPORTE

**BRENNO CURTOLO CAVALCANTI
GABRIEL PERES DA SILVA
LUIIS ARTHUR RABELO BARBOSA
RAFAEL HENRIQUE GUIMARÃES**

COMPUTAÇÃO CIENTÍFICA E OTIMIZAÇÃO



O PROBLEMA DO TRANSPORTE

- Distribuição de recursos (como mercadorias) de várias fontes para vários destinos.
- O objetivo é determinar a quantidade de recursos a serem enviados de cada fonte para cada destino, de modo a minimizar os custos totais de transporte.



Aplicações

A empresa de jogos "Fantasy Games" está lançando o tão aguardado jogo "God of War" em três diferentes regiões. Existem três fornecedores de jogos, porém um deles acabou não recebendo nenhuma unidade, e três lojas de vendas nessas regiões. A empresa precisa planejar a distribuição dos jogos entre os fornecedores e as lojas para atender à demanda esperada. A matriz de ofertas e demandas é fornecida abaixo:

Capacidade de envio dos centros de fornecimento:

Centro de Fornecimento A: 40 unidades

Centro de Fornecimento B: 30 unidades

Centro de Distribuição C: 0 unidades

Demandas diárias de produtos nos destinos:

Loja A: 25 unidades

Loja B: 35 unidades

Loja C: 60 unidades



MÉTODO DO CANTO NOROESTE

Se a oferta na linha atual for menor ou igual à demanda na coluna atual, então o código calcula o custo multiplicando a oferta pelo custo correspondente e subtrai a oferta da demanda. Em seguida, ele avança para a próxima linha.

Se a oferta for maior que a demanda, o código calcula o custo multiplicando a demanda pelo custo correspondente e subtrai a demanda da oferta. Em seguida, ele avança para a próxima coluna.



Método do Canto Noroeste

Solução Inicial

		Destinos	1	2	3	4	Oferta
Origens		1	8	6	10	9	35
		2	9	12	13	7	50
1	35						35
2	10	20	20				50
3	14	9	16	10	30	5	40
	Demandas	45	20	30	30		

MÉTODO DO CANTO NOROESTE

Método do Canto Noroeste

Solução Inicial

Destinos \ Origens	1	2	3	4	Oferta
Origens	8	6	10	9	35
1	35				
2	9	12	13	7	50
3	14	9	16	5	40
Demandas	45	20	30	30	

DEF NOROESTE(CUSTOS, OFERTA, DEMANDA):

ANS = 0

LINHA = 0

COLUNA = 0

BFS = []

WHILE(LINHA != LEN(CUSTOS) AND COLUNA != LEN(CUSTOS[0])):

IF (OFERTA[LINHA] <= DEMANDA[COLUNA]):

ANS += OFERTA[LINHA] * CUSTOS[LINHA][COLUNA];

DEMANDA[COLUNA] -= OFERTA[LINHA];

BFS.APPEND(((LINHA, COLUNA), OFERTA[LINHA]));

LINHA += 1;

ELSE:

ANS += DEMANDA[COLUNA] * CUSTOS[LINHA][COLUNA];

OFERTA[LINHA] -= DEMANDA[COLUNA];

BFS.APPEND(((LINHA, COLUNA), DEMANDA[COLUNA]));

COLUNA += 1;

RETURN ANS, BFS

MÉTODO DO CANTO NOROESTE

Método do Canto Noroeste

Solução Inicial

Destinos \ Origens	1	2	3	4	Oferta
Origens	8	6	10	9	35
1	35				
2	9	12	13	7	50
3	14	9	16	5	40
Demandas	35	30	30	30	

DEF NOROESTE(CUSTOS, OFERTA, DEMANDA):

ANS = 0

LINHA = 0

COLUNA = 0

BFS = []

WHILE(LINHA != LEN(CUSTOS) AND COLUNA != LEN(CUSTOS[0])):

IF (OFERTA[LINHA] <= DEMANDA[COLUNA]):

ANS += OFERTA[LINHA] * CUSTOS[LINHA][COLUNA];

DEMANDA[COLUNA] -= OFERTA[LINHA];

BFS.APPEND(((LINHA, COLUNA), OFERTA[LINHA]));

LINHA += 1;

ELSE:

ANS += DEMANDA[COLUNA] * CUSTOS[LINHA][COLUNA];

OFERTA[LINHA] -= DEMANDA[COLUNA];

BFS.APPEND(((LINHA, COLUNA), DEMANDA[COLUNA]));

COLUNA += 1;

RETURN ANS, BFS

MÉTODO DO CANTO NOROESTE

RELAÇÃO INICIAL						
	Custos/Ofertas/Demandas					
	Fornecedor C	A	B	C	Oferta	
Fornecedor A	500	50	10	80	40	
Fornecedor B	30	100	120	30	30	
Fornecedor C	0	40	20	500	120	
Demandada	120	25	35	60	240/190	



MÉTODO DO CANTO NOROESTE

NOROESTE						
Custos/Ofertas/Demandas						
	Fornecedor C	A	B	C	Oferta	
Fornecedor A	500	50	10	80	40	
Fornecedor B	30	100	120	30	30	
Fornecedor C	0	40	20	500	120	
Artificial	0	0	0	0	0	50
Demandada	120	25	35	60	240/240	



MÉTODO DO CANTO NOROESTE

	Fornecedor C	Loja A	Loja B	Loja C	u	Oferta	
Fornecedor A	500	50	10	80		40	
	40						
Fornecedor B	30	100	120	30		30	
	30						
Fornecedor C	0	40	20	500		120	
	50	25	35	10			
Artificial	0	0	0	0		50	
				50			
v							
Demandas	120	25	35	60	0		

SBF INICIAL: X0 = (25 , 15, 0, 0, 0, 20, 10, 0, 0, 0, 50, 70);

$$Z_0 = ((500 * 40) + (30 * 30) + (50 * 0) + (25 * 40) + (35 * 20) + (500 * 10) + (50 * 0))$$

$$Z_0 = ((20.000) + (900) + (0) + (1000) + (700) + (5000) + (0)) = 27600$$

MÉTODO DO MÍNIMO DOS CUSTOS

Canva

A variável básica escolhida é a variável que corresponde ao menor custo (em caso de empate a escolha é arbitrária).

A primeira variável básica escolhida será sempre a de menor custo, depois será escolhida como variável básica a de menor custo no quadro resultante relativo ao que foi traçado, e assim sucessivamente, até terem sido traçadas todas as linhas e todas as colunas.



MÉTODO DO MÍNIMO DOS CUSTOS

Canva

	Fornecedor C	Loja A		Loja B		Loja C		u	Oferta
Fornecedor A	500	50		10		80		40	
Fornecedor B	30	100		120		30		30	
Fornecedor C	0	40		20		500		0	
Artificial	120								
v	0	0		0		0		50	
Demanda	0	25		35		60		0	



MÉTODO DO MÍNIMO DOS CUSTOS

Canva

	Fornecedor C	Loja A	Loja B	Loja C	u	Oferta
Fornecedor A	500	50	10	80		40
Fornecedor B	30	100	120	30		30
Fornecedor C	0	40	20	500		120
Artificial	120	0	0	0		25
v		25				
Demanda	120	0	35			0



MÉTODO DO MÍNIMO DOS CUSTOS

Canva

	Fornecedor C	Loja A		Loja B		Loja C		u	Oferta
Fornecedor A	500	50	10	80				40	
Fornecedor B	30	100	120	30				30	
Fornecedor C	0	40	20	500				0	
Artificial	120							0	
v	0	0	0	0				0	
Demanda	0	0	10	60				0	



MÉTODO DO MÍNIMO DOS CUSTOS

Canva

	Fornecedor C	Loja A		Loja B		Loja C		u	Oferta
Fornecedor A	500	50		10		80			30
Fornecedor B	30	100		120		30			30
Fornecedor C	0	40		20		500			0
Artificial	120								0
V	0	0		0		0			0
Demanda	0	0		0		30			0



MÉTODO DO MÍNIMO DOS CUSTOS

Canva

	Fornecedor C	Loja A	Loja B	Loja C	u	Oferta
Fornecedor A	500	50	10	80		30
Fornecedor B	30	100	120	30		0
Fornecedor C	0	40	20	500		0
Artificial	120					0
V	0	0	0	0		0
Demandado	0	0	0	0		0



MÉTODO DO MÍNIMO DOS CUSTOS

Canva



```
def minimo_custo(custos, oferta, demanda):
    ans = 0
    bfs = []
    INF = 10**3
    n = len(custos)
    m = len(custos[0])
    while max(oferta) != 0 or max(demanda) != 0:
        mini1 = INF
        for i in range(n):
            for j in range(m):
                if (custos[i][j] < mini1 and oferta[i] > 0 and
                    demanda[j] > 0):
                    mini1 = custos[i][j]
                    ind1, ind2 = i, j
        mini2 = min(oferta[ind1], demanda[ind2])
        bfs.append(((ind1, ind2), mini2))
        ans += mini2 * mini1
        oferta[ind1] -= mini2
        demanda[ind2] -= mini2
        custos[ind1][ind2] = INF
    return ans, bfs
```



MÉTODO DE VOGEL

Trabalha analisando a MAIOR MENOR diferença entre as linhas e as colunas.

Se a maior diferença for na linha, o código encontra o menor custo na linha correspondente e calcula a oferta e demanda mínimas para essa célula. O custo é adicionado à resposta, as ofertas e demandas são atualizadas e a linha ou coluna correspondente é marcada como "eliminada" atribuindo valores altos a essas células. Faz o mesmo processo com colunas.

1	2	3	4	1	6
4	3	2	4	1	8
0	2	2	1	1	3
1	0	0	3		
4	7	6	7		

mínimo

máximo

MÉTODO DE VOGEL

```
def findDiff(custos):
    linhaDiff = []
    colunaDiff = []
    for i in range(len(custos)):
        arr = custos[i][:]
        arr.sort()
        linhaDiff.append(arr[1]-arr[0])
    coluna = 0
    while coluna < len(custos[0]):
        arr = []
        for i in range(len(custos)):
            arr.append(custos[i][coluna])
        arr.sort()
        coluna += 1
        colunaDiff.append(arr[1]-arr[0])
    return linhaDiff, colunaDiff
```

Essa função calcula as diferenças entre os dois menores valores em uma linha e em uma coluna.

MÉTODO DE VOGEL

Vogel		Custos/Ofertas/Demandas				
	Fornecedor C	A	B	C	Oferta	
Fornecedor A	500	50	10	80	40	
Fornecedor B	30	100	120	30	30	
Fornecedor C	0	40	20	500	120	
Artificial	30	0	0	0	50	
Demanda	120	25	35	60	240/240	

	Matriz de Custos					
	F.c	A	B	C	Oferta	Diferenças
	500	50	10	80		
	30	100	120	30		
	0	40	20	500		
	0	0	0	0		
Demandas						
Diferenças						



MÉTODO DE VOGEL

	4	7	8	9			
	Matriz de Custos						
	F.c	A	B	C	Oferta	Diferenças	
	500	50	10	80	40	40	
	30	100	120	30	30	0	
	0	40	20	500	120	20	
	0	0	0	0	50	0	
Demanda	120	25	35	60			
Diferenças	0	40	10	30			

	Matriz de Custos						
	F.c	A	B	C	Oferta	Diferenças	
	500	50	10 (35)	80	40	40	
	30	100	120	30	30	0	
	0	40	20	500	120	20	
	0	0	0	0	50	0	
Demanda	120	25	35	60			
Diferenças	0	40	10	30			

MÉTODO DE VOGEL

	Matriz de Custos						
	F.c	A	B	C	Oferta	Diferenças	
	500	50 (5)	10	80	5	70	
	30	100	120	30	30	0	
	0	40	20	500	120	20	
	0	0	0	0	50	0	
Demanda	120	25	0	60			
Diferenças	0	40	0	30			

	Matriz de Custos						
	F.c	A	B	C	Oferta	Diferenças	
	500	50	80	0			
	30	100	120	30	30	0	
	0 (120)	40	20	500	120	40	
	0	0	0	0	50	0	
Demanda	120	20	0	60			
Diferenças	0	40	0	30			

MÉTODO DE VOGEL

	Matriz de Custos					
	F.c	A	B	C	Oferta	Diferenças
	500	50		80	0	
	30	100	120	30 (30)	30	70
	0	40		500	0	
	0	0	0	0	50	0
Demandas	0	20	0	60		
Diferenças		40	0	30		

	Matriz de Custos					
	F.c	A	B	C	Oferta	Diferenças
	500	50		80	0	
	30	100	120	30	0	
	0	40		500	0	
	0	0	0	0 (30)	50	0
Demandas	0	20	0	30		
Diferenças		0	0	0		

MÉTODO DE VOGEL

	Matriz de Custos					
	F.c	A	B	C	Oferta	Diferenças
	500	50		80	0	
	30	100	120	30	0	
	0	40	20	500	0	
	0	0 (20)	0	0 (30)	20	0
Demandas	0	20	0	0		
Diferenças		0	0			

	Matriz de Custos					
	F.c	A	B	C	Oferta	Diferenças
	500	50		80	0	
	30	100	120	30	0	
	0	40	20	500	120	
	0	0	0	0	20	
Demandas	0	0	0	0		
Diferenças		0				
					FIM	

MÉTODO DE VOGEL

Passo a Passo:							u	Oferta	
	Fornecedor C	Loja A		Loja B		Loja C			
Fornecedor A	500	50	10	80					40
		5 2 ^º	35 1 ^º						
Fornecedor B	30	100	120	30					30
				30 4 ^º					
Fornecedor C	0	40	20	500					120
	120 3 ^º								
Artificial	0	0	0	0					50
		20 6 ^º		30 5 ^º					
v									
Demandas	120	25	35	60					0



SOLUÇÃO ÓTIMA

× × Canva × ×

Esta é a função principal que implementa o método simplex para resolver o problema de transporte. Ela recebe a matriz custos contendo os custos das posições e a matriz bfs que contém as posições básicas e os respectivos valores.

```
def transportation_simplex_method(custos, bfs):
    def inner(bfs):
        us, vs = get_us_and_vs(bfs, custos)
        ws = get_ws(bfs, custos, us, vs)
        if can_be_improved(ws):
            ev_position = get_entering_variable_position(ws)
            loop = get_loop([p for p, v in bfs], ev_position)
            return inner(loop_pivoting(bfs, loop))
        return bfs

    variaveis_basicas = inner(bfs)
    solucao = np.zeros((len(custos), len(custos[0])))
    for (i, j), v in variaveis_basicas:
        solucao[i][j] = v

    return solucao
```



SOLUÇÃO ÓTIMA

× × Canva × ×

Essa função calcula os valores das variáveis us e vs usando o método de cálculo de custos. Ela recebe a matriz bfs que contém as posições básicas e os respectivos valores, e a matriz custos que contém os custos das posições. A função retorna as listas us e vs, onde us contém os valores calculados para as linhas e vs contém os valores calculados para as colunas.



```
def get_us_and_vs(bfs, custos):
    us = [None] * len(custos)
    vs = [None] * len(custos[0]))
    us[0] = 0
    bfs_copy = bfs.copy()
    while len(bfs_copy) > 0:
        for index, bv in enumerate(bfs_copy):
            i, j = bv[0]
            if us[i] is None and vs[j] is None:
                r = 0
                for index, bv in enumerate(bfs_copy):
                    a,b = bv[0]
                    if us[a] is not None or vs[b] is not None:
                        r = 1
                if r == 0:
                    if(i+1<len(custos)):
                        us[i+1] = 0
                        vs[j] = custos[i+1][j]
                        bfs_copy.append(((i+1, j),0))
                        bfs.append(((i+1, j),0))
                elif(i-1>0):
                    us[i-1] = 0
                    vs[j] = custos[i-1][j]
                    bfs_copy.append(((i-1, j),0))
                    bfs.append(((i-1, j),0))
                r = 0
                continue
            custo = custos[i][j]
            if us[i] is None:
                us[i] = custo - vs[j]
            else:
                vs[j] = custo - us[i]
            bfs_copy.pop(index)
            break
    return us, vs
```

SOLUÇÃO ÓTIMA

× × Canva × ×

Esta função calcula os valores das variáveis ws (variáveis de trabalho) para determinar se a solução atual é ótima ou se pode ser melhorada. Ela recebe a matriz bfs, as matrizes custos, us e vs. A função retorna uma lista de tuplas, onde cada tupla contém a posição (i, j) e o valor $us[i] + vs[j] - custos[i][j]$ para as posições não básicas.

```
def get_ws(bfs, custos, us, vs):
    ws = []
    for i, linha in enumerate(custos):
        for j, custo in enumerate(linha):
            non_basic = all([p[0] != i or p[1] != j for p, v in bfs])
            if non_basic:
                ws.append(((i, j), us[i] + vs[j] - custo))

    return ws
```



SOLUÇÃO ÓTIMA

× × Canva × ×

Esta função constrói um loop a partir de uma posição de variável de entrada (entering variable) e uma lista de posições das variáveis básicas (basic variables). Ela usa uma função interna inner para construir recursivamente o loop. A função retorna o loop construído como uma lista de posições (i, j).

```
def get_loop(bv_positions, ev_position):
    def inner(loop):
        if len(loop) > 3:
            can_be_closed = len(get_possible_next_nodes(loop, [ev_position])) == 1
            if can_be_closed: return loop

            not_visited = list(set(bv_positions) - set(loop))
            possible_next_nodes = get_possible_next_nodes(loop, not_visited)
            for next_node in possible_next_nodes:
                new_loop = inner(loop + [next_node])
                if new_loop: return new_loop

    return inner([ev_position])
```



SOLUÇÃO ÓTIMA

× × Canva × ×

Esta função executa o pivoteamento no loop para obter uma nova base factível. Ela recebe a matriz bfs que contém as posições básicas e os respectivos valores, e o loop que é uma lista de posições (i, j). A função determina a posição de saída (leaving position) e o valor de saída (leaving value) no loop, e realiza as operações de pivoteamento para obter uma nova matriz bfs atualizada.

```
def loop_pivoting(bfs, loop):
    even_cells = loop[0::2]
    odd_cells = loop[1::2]
    get_bv = lambda pos: next(v for p, v in bfs if p == pos)
    leaving_position = sorted(odd_cells, key=get_bv)[0]
    leaving_value = get_bv(leaving_position)

    new_bfs = []
    for p, v in [bv for bv in bfs if bv[0] != leaving_position] + [(loop[0], 0)]:
        if p in even_cells:
            v += leaving_value
        elif p in odd_cells:
            v -= leaving_value
        new_bfs.append((p, v))

    return new_bfs
```



CONTROLE DE DADOS

Canva

Tabela de oferta e demanda:

	Oferta	Demanda
Fornecedor A	40	
Fornecedor B	30	
Fornecedor C	0	0
Loja A		25
Loja B		35
Loja C		60

Tabela de custos:

Envio	Chegada	Custo
Fornecedor A	Loja A	50
Fornecedor A	Loja B	10
Fornecedor A	Loja C	80
Fornecedor B	Loja A	100
Fornecedor B	Loja B	120
Fornecedor B	Loja C	30
Fornecedor B	Fornecedor C	30
Fornecedor C	Loja A	40
Fornecedor C	Loja B	20

Tabela final:

	Fornecedor C	Loja A	Loja B	Loja C	Oferta
Fornecedor A	500	50	10	80	40
Fornecedor B	30	100	120	30	30
Fornecedor C	0	40	20	500	120
Artificial	0	0	0	0	50
Demanda	120	25	35	60	0



