UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA QUÍMICA

OTIMIZAÇÃO DE PROCESSOS INDUSTRIAIS: MODELAGEM DE PROBLEMAS DE PROGRAMAÇÃO LINEAR E NÃO-LINEAR

Paulo Eduardo Reis Maia

Trabalho de Graduação apresentado ao departamento de Engenharia Química da Universidade Federal de São Carlos

Orientador: Prof. Felipe Fernando Furlan

São Carlos – SP

2022

BANCA EXAMINADORA

Trabalho de Graduação apresentado no dia 12/09 perante a seguinte banca examinadora:

Orientador: Prof. Felipe Fernando Furlan, DEQ - UFSCar

Convidado: Prof. Antonio Carlos Luperni Horta, DEQ - UFSCar

Professor da Disciplina: Prof. Ruy de Sousa Júnior, DEQ – UFSCar

AGRADECIMENTOS

Agradeço a todos que me acompanharam ao longo dessa jornada que agora se encerra, pela parceria e pelos momentos compartilhados. Aos meus pais, pelas oportunidades. À minha irmã e primos, e a toda minha família; às minhas avós, que viram o começo disso tudo, e talvez, quem sabe, estejam vendo agora o final. Aos amigos de Minas, que estiveram lá desde sempre, especialmente os do Congresso e umas certas loiras aí; aos de Ribeirão e do estágio; e aos de São Carlos, dos almoços e das sucadas, da AEQ e *do* AEQ - *só os inomináveis*.

Agradeço por todo o suporte neste trabalho, e também pela amizade, ao meu professor orientador, Felipe; ao grupo de estudos de otimização; e ao Geraldo, que me deu a visão além do alcance aos quarenta e cinco do segundo tempo.

Agradeço o apoio e patrocínio de todos que humildemente me cederam uma bananinha para fazer review ou mandaram o pix.

“Aqueles que não têm determinação são incapazes até de enxugar as próprias lágrimas.”

- Satsuki Kiryuin

RESUMO

A eficiência de um processo químico industrial está atrelada a uma série de variáveis, que não se limitam apenas às variáveis de processo, mas também compreende fatores externos, como custo de matéria-prima, transporte, armazenamento, entre outros. O conceito de otimização pode ser aplicado a diferentes aspectos de um processo, sendo comumente empregado na minimização de custos totais ou tempo de ciclo; maximização dos lucros ou da produção, tendo em vista a capacidade da planta; e até mesmo na definição da localização onde uma unidade produtiva será construída. Fazer esta análise é vital para garantir a viabilidade de um projeto, até mesmo os de menor escala, e esta preocupação é exacerbada pelo contexto geopolítico dos dias de hoje, no qual a redução do desperdício e do consumo de recursos naturais tornou-se uma prioridade na busca por formas de produção mais sustentáveis. Neste trabalho, dois estudos de caso de problemas de engenharia foram abordados, sendo modelados e solucionados utilizando Pyomo, um pacote do Python 3 com linguagem de modelagem própria focado em otimização de processos. Os resultados obtidos, quando comparados aos da literatura, se mostraram muito semelhantes e satisfatórios, e serão disponibilizados, juntamente com os problemas, em um acervo que ficará disponível aos alunos do curso de Engenharia Química, e poderá ser empregado como ferramenta de estudo na disciplina de Síntese e Otimização de Processos Químicos.

ABSTRACT

The efficiency of industrial chemical processes is tied not only to process variables, but also to external factors, such as the availability and cost of raw materials, transportation, storage, and many others. The concept of optimization may be applied to various aspects of the same process, and it is most utilized in minimizing production costs or lead time; maximizing profits or production volume; and even defining the layout of facilities or choosing the most appropriate location for a building site, according to its proximity to suppliers and clients. Conducting optimization analysis is essential to ensure that even smaller projects are viable, even more so now that reducing waste and using our natural resources more efficiently have become priorities in the search for more sustainable manufacturing. In this paper, two engineering problems were modeled in Pyomo, an open-source optimization Python package. The results were very close to those found in the articles from which the problems were extracted, and will be compiled, along with the developed models, for further study in the Chemical Process Optimization and Synthesis course, as part of the Chemical Engineering curriculum.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS E QUADROS

LISTA DE TABELAS

1 - INTRODUÇÃO E OBJETIVOS

Otimização matemática consiste em encontrar uma solução ótima, dentre um número de soluções possíveis, para um determinado problema, de acordo com um critério específico. O caso mais tradicional de aplicação da otimização é a minimização ou maximização de uma função objetivo. Para tal, as variáveis e parâmetros desta função são variadas, e os valores que determinam o ótimo da função objetivo são escolhidos mediante um algoritmo. Este tem sido um tópico de interesse desde a Era Moderna, e célebres matemáticos desenvolveram métodos para aplicação de otimização, como os multiplicadores de Lagrange e os algoritmos de Newton e Gauss. Uma classe de problemas de otimização que se mostrou de grande importância no decorrer do século XX são os problemas de programação linear.

A formulação de problemas de programação linear (*linear programming* ou LP) foi desenvolvida independentemente por estudiosos ao longo da Segunda Guerra Mundial, primeiro em 1939, pelo economista soviético Leonid Kantorovich, e pelo matemático e físico americano Frank L. Hitchcock em 1941. Estas formulações visavam a otimização de custos e planejamento de gastos do exército, e a maximização das baixas incorridas pelos adversários. As soluções propostas nessa época, especialmente por Hitchcock, se assemelhavam muito ao posterior método Simplex, desenvolvido por George B. Dantzig, entre 1946 e 47.

O método Simplex se destacou por ser o primeiro método que abordava eficientemente a maior parte dos problemas de programação linear relevantes na época, e continuou a ser aplicado no meio industrial nas décadas seguintes. Embora hoje em dia, disponhamos de outros métodos mais eficientes e com áreas de aplicação mais amplas, a formulação de problemas como programação linear ainda é válida e eficaz em diversas situações no contexto da produção industrial, e seu estudo se mostra muito relevante e interessante na formação de novos profissionais do meio.

Na modelagem de um problema como programação linear, as funções objetivo e de restrição são funções lineares. Um problema de programação linear inteira mista (*mixed-integer linear programming*, ou MILP) é um caso particular de LP, na qual algumas das variáveis envolvidas assumem valores inteiros, enquanto outras não. Quando todas as variáveis de um problema LP assumem valores inteiros, o problema é dito de programação inteira (*integer linear programming*). É comum o uso de variáveis inteiras para representar quantidades indivisíveis (como o número de itens a serem produzidos) ou decisões binárias (em que a variável é Booleana, e só pode assumir o valor de 0 ou 1).

Neste trabalho, será apresentada uma revisão bibliográfica da literatura, e um estudo aprofundado de dois exemplos de problemas reais, um de programação linear inteira mista e outro de programação não-linear, resolvidos usando bibliotecas disponíveis na linguagem de programação Python. A eficiência dos algoritmos utilizados será comparada com a dos métodos utilizados na resolução do problema original, disponíveis na literatura. Os resultados serão compilados e ficarão disponíveis como material auxiliar para estudo aos alunos do curso de Engenharia Química da UFSCar, na disciplina de Síntese e Otimização de Processos, visando possibilitar maior oportunidade de contato com métodos computacionais e enriquecimento do aprendizado.

2 - REVISÃO BIBLIOGRÁFICA

O conceito de otimização está fundamentado em três pilares: a função objetivo, as funções de restrição (normalmente de desigualdade) e o modelo do processo (restrições de igualdade) (SECCHI e BISCAIA, 2012). A função objetivo representa o aspecto do processo que se deseja otimizar, isto é, minimizar ou maximizar. A correta formulação de uma função objetivo é crucial para a aplicação eficiente de um método de otimização, uma vez que são as características dela que parcialmente determinam o algoritmo a ser escolhido, e um aumento em sua complexidade requer o uso de algoritmos mais robustos, tornando o processo mais custoso.

A complexidade de uma função objetivo está relacionada com a dimensão do problema, isto é, o número de variáveis que a compõem, e também às não-linearidades das funções envolvidas (objetivo, restrições de igualdade e desigualdade). Durante a formulação, é imprescindível atentar-se às possibilidades de simplificação desta função, tendo em vista os parâmetros utilizados e o contexto do problema real, para garantir um nível de complexidade que o descreva satisfatoriamente e garanta o mínimo custo computacional necessário. Da mesma forma, a presença de restrições e as características das funções que as descrevem também influenciam na escolha do algoritmo a ser empregado, e juntamente com a função objetivo, elas compõem o modelo do processo que se pretende otimizar.

2.1 – Otimização sem restrições

Como o próprio nome sugere, um problema de otimização sem restrições apresenta um modelo de processo composto por uma única equação, isto é, a função objetivo. Os métodos para solucionar problemas irrestritos podem ser categorizados quanto ao uso ou não da derivada da função objetivo.

A inclusão da derivada nos cálculos é característica dos métodos indiretos e métodos analíticos, tendo sido estes os primeiros desenvolvidos. Quando comparados aos métodos que não utilizam derivada – métodos de busca e métodos diretos – os métodos analíticos convergem mais rapidamente, no entanto tem como desvantagem o requerimento da continuidade da função objetivo e de sua derivada (SECCHI e BISCAIA, 2012).

Com o advento de computadores mais modernos, capazes de realizar cálculos complexos com maior velocidade, os métodos analíticos foram suplantados por métodos numéricos iterativos (EDGAR, HIMMELBLAU e LASDON, 2001).

O estudo de problemas irrestritos envolvendo uma função objetivo de uma única variável é de grande importância, apesar de ser um dos problemas de menor complexidade. De fato, diversos problemas são univariados ou podem ser formulados assim após manipulações matemáticas. Além disso, as técnicas de busca empregadas na resolução de problemas unidimensionais compõe diversos algoritmos de resolução de problemas multidimensionais, tanto irrestritos, quanto restritos (EDGAR, HIMMELBLAU e LASDON, 2001). Nesses métodos, após a definição da direção de busca, é realizada uma busca em linha, ou seja, uma otimização unidimensional na direção determinada anteriormente.

Dentre os métodos que empregam a técnica de busca em linha para solução de problemas com múltiplas variáveis, destacam-se os métodos que utilizam apenas os valores da função objetivo e de sua primeira derivada, e os métodos de Newton e quasi-Newton. De forma geral, estes métodos iterativos partem ou de um valor (“chute”) inicial da função objetivo, ou de valores iniciais para condições de contorno do problema, e a cada iteração, repetem uma sequência de dois passos: escolha de uma direção de busca, seguida da minimização do valor da função nesta direção. A principal diferença entre cada um dos métodos está na forma com que é determinada a direção de busca (EDGAR, HIMMELBLAU e LASDON, 2001).

2.2 – Condições de otimalidade

A maneira mais fácil de analisar as condições para que dado ponto *x** seja candidato a ponto de mínimo ou máximo local de uma função *f(x)* é fazer uma expansão em série de Taylor ao redor deste ponto, truncada no termo de segunda ordem, como na equação 2.1.

$$f(x) = f(x^*) + \nabla^T f(x^*)\,\Delta x + \frac{1}{2}(\Delta x^T)\,\nabla^2 f(x^*)\,\Delta x \qquad (2.1)$$

Suponhamos que *x** seja mínimo local da função *f(x)*. Neste caso, a diferença entre *f(x)* e *f(x*)*, como demonstrado na equação 2.2, deve ser sempre positiva ou nula, dado que *f(x*)* é o menor valor que a função pode assumir naquela vizinhança. Quando isto é verdade para qualquer valor de *x* no espaço em questão, dizemos que *x** é mínimo local, e quando isso se aplica a todo o espaço, mínimo global.

$$f(x) - f(x^*) = \nabla^T f(x^*)\,\Delta x + \frac{1}{2}(\Delta x^T)\,\nabla^2 f(x^*)\,\Delta x = 0 \qquad (2.2)$$

Observemos agora o primeiro termo do lado direito da equação (2.2). $\Delta x = x - x^*$ pode assumir tanto valores positivos, quanto negativos. Portanto, para que o valor de *f(x)* seja sempre igual ou superior a *f(x*)* na vizinhança desse último, o gradiente da função objetivo deve ser nulo, desconsiderando a princípio a contribuição do termo de segunda ordem. Isso nos dá a condição de otimalidade de primeira ordem: $\nabla f(x) = 0$.

Já no segundo termo, o gradiente de segunda ordem da função corresponde à matriz hessiana *H(x),* que multiplica o produto entre o vetor $\Delta x = x - x^*$ e seu transposto. Como a matriz Hessiana é simétrica, seus autovetores serão ortogonais entre si, formando uma base para o espaço n-dimensional no qual as variáveis independentes estão. Dessa forma, qualquer vetor *x* poderá ser expresso como uma combinação linear dos autovetores (v_i) de *H*. Assim, o segundo termo da expansão pode ser escrito como na equação 2.3:

$$f(x) - f(x^*) = \Delta x^T H \Delta x = \varepsilon^T V^T H V \varepsilon = \sum_{i=1}^{n} \lambda_i\, \varepsilon_i^2 |v_i|^2 \qquad (2.3)$$

Onde V é a matriz de autovetores de H, λ são os autovalores e ε as coordenadas de Δx escritas na base V. Deste modo, o valor do lado esquerdo da equação depende unicamente do valor dos autovalores da matriz hessiana. Quando estes forem todos superiores a zero, dizemos que a matriz é positiva definida, e corresponde a um ponto de mínimo. Analogamente, quando a matriz é negativa definida, o ponto encontrado é de máximo. Assim, a condição necessária de otimalidade de segunda ordem para a existência de um ponto de mínimo é que a matriz hessiana seja positiva semidefinida, conforme equação 2.4:

$$\Delta x^T H \Delta x \geq 0 \tag{2.4}$$

Além disso, temos a condição suficiente para a existência de um ponto de mínimo, que advém da junção das duas condições de otimalidade, representadas pelas equações 2.5 e 2.6:

$$\nabla f(x) = 0 \tag{2.5}$$

$$\Delta x^T H \Delta x > 0 \tag{2.6}$$

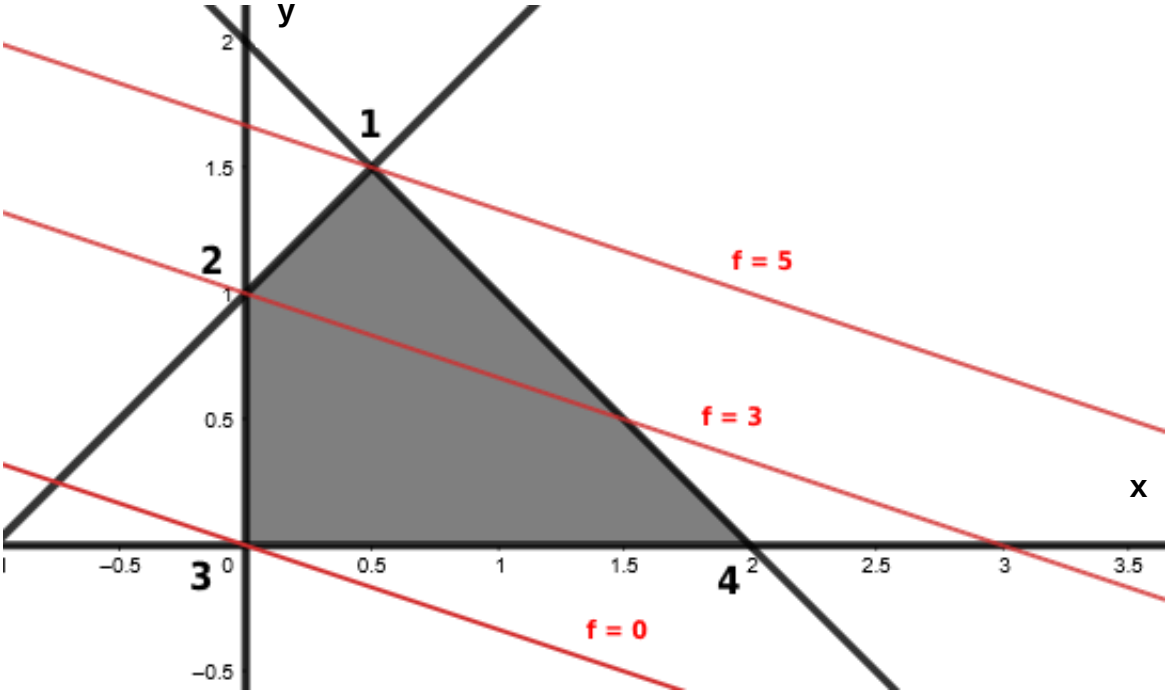
2.3 – Otimização com restrições

Um problema de otimização pode estar sujeito a duas categorias de restrições, sendo estas chamadas restrições de igualdade ou desigualdade, modeladas respectivamente por equações e inequações. No contexto do problema real, estas restrições podem representar impossibilidades físicas, limites de capacidade ou metas a serem atingidas.

Uma restrição é dita ativa em um ponto viável quando assume valor nulo naquele ponto. Deste modo, restrições de igualdade são ativas em qualquer ponto viável, enquanto que restrições de desigualdade podem ou não ser ativas em dado

ponto. Isto é ilustrado no exemplo do sistema de equações 2.7, adaptado de EDGAR, HIMMELBLAU e LASDON (2001).

Figura 2.1. Representação gráfica de um problema linear



Fonte: Adaptado de Edgar, Himmelblau e Lasdon (2001).

$$\begin{aligned} &\max\ f = x_1 + 3\,x_2 \\ &sujeito\ a: \\ &-x_1 + x_2 \leq 1 \\ &x_1 + x_2 \leq 2 \\ &x_1, x_2 \geq 0 \end{aligned} \qquad (2.7)$$

O problema em questão apresenta quatro restrições de desigualdade. A região viável está delimitada pelas retas que representam estas funções, na área interna à região destacada. O exemplo acima consiste em um problema de

programação linear, onde a função objetivo e as restrições são lineares. Este texto focará nos problemas de programação linear.

Em um espaço de *n* dimensões, os pontos extremos da função sempre ocorrem em vértices da região viável. Estes vértices são determinados pela intersecção de ao menos *n* restrições ativas (que em duas dimensões, correspondem a retas). Pode haver restrições redundantes em um mesmo vértice, isto é, número de restrições excede n (EDGAR, HIMMELBLAU e LASDON, 2001).

Atribuindo valores constantes à função objetivo *f* (representados, na figura 2.1, pelas paralelas destacadas, que correspondem a *f* = 0, *f* = 3 e *f* = 5) podemos encontrar o ótimo da função na intersecção de uma destas paralelas com um vértice. No exemplo, como queremos maximizar o valor de *f*, este ótimo se encontra no vértice *1*. É possível que mais de um vértice corresponda ao valor ótimo da função; neste caso, ela tem múltiplas soluções viáveis.

Há ainda casos em que a região viável não está totalmente delimitada, e a função objetivo é ilimitada. Por outro lado, o conjunto de restrições pode delimitar uma região vazia, para qual não há solução.

De acordo com a natureza do modelo, os problemas de otimização com restrições podem ser classificados como programação linear, não-linear, quadrática, inteira ou inteira mista (SECCHI e BISCAIA, 2012). Neste contexto, o termo “programação” não se refere a programação computacional, mas é usado como sinônimo de otimização. O presente trabalho tem como foco problemas de programação linear, que apresentam função objetivo e restrições lineares, e podem ser definidos conforme o sistema de equações 2.8:

$$\begin{aligned} &\min S(x) = c^T x \\ &sujeito\ a: \\ &A\,x \leq b \\ &x \geq 0 \end{aligned} \tag{2.8}$$

Sendo *A* uma matriz *m x n*, temos um problema com *m* restrições e *n* variáveis.

A programação linear é dita uma programação convexa: a função objetivo é convexa e as restrições formam um conjunto convexo. Isto determina que o extremo local seja também global (SECCHI e BISCAIA, 2012).

2.4 – Método Simplex

O método Simplex é um procedimento aplicado na solução de problemas de programação linear, que consiste em duas etapas. Na primeira, é obtida uma solução básica viável para o problema, se existir uma. Neste caso, procede-se para a segunda etapa, que utiliza esta solução como ponto de partida para buscar uma solução que minimize a função objetivo. Em caso negativo, pode-se inferir que as restrições do modelo são inconsistentes (EDGAR, HIMMELBLAU e LASDON, 2001).

O algoritmo do Simplex se inicia com um sistema de equações expandido, que consiste nas restrições de igualdade, juntamente com a função objetivo. Quando o problema apresenta restrições de desigualdade, faz-se necessária a transformação destas em igualdades através da introdução das chamadas variáveis de folga, conforme as equações 2.9 e 2.10 (SECCHI e BISCAIA, 2012).

$$A\,x \leq b \rightarrow A\,x + f = b, \quad b \geq 0 \tag{2.9}$$

$$A\,x \geq b \rightarrow A\,x - f = b, \quad b \geq 0 \tag{2.10}$$

Em um problema de programação linear, a função objetivo (equação 2.11) pode ser reescrita como a equação 2.12, que será integrada ao sistema das equações de restrição.

$$f = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \tag{2.11}$$

$$-f + c_1 x_1 + c_2 x_2 + \cdots + c_n x_n = 0 \tag{2.12}$$

O sistema resultante 2.13, composto por *m* equações e *n* + *m* variáveis x_{ij} deve ser reduzido à forma canônica, isto é, a variável x_{ij} deve ter coeficiente unitário na equação *i* e coeficiente nulo nas demais (DANTZIG e THAPA, 1997). Para tal, é necessário especificar n variáveis não-básicas ($x_{m+1,}$ ..., x_n) sendo as demais (variáveis básicas) obtidas em função destas (SECCHI e BISCAIA, 2012).

$$\begin{gathered} x_1 + \overline{a}_{1,m+1}\, x_{m+1} + \cdots + \overline{a}_{1,n}\, x_n = \overline{b}_1 \\ x_2 + \overline{a}_{2,m+1}\, x_{m+1} + \cdots + \overline{a}_{2,n}\, x_n = \overline{b}_2 \\ \vdots \\ x_m + \overline{a}_{m,m+1}\, x_{m+1} + \cdots + \overline{a}_{m,n}\, x_n = \overline{b}_1 \\ -f + \overline{c}_{m+1} x_{m+1} + \cdots + \overline{c}_n x_n = -\overline{f} \end{gathered} \qquad (2.13)$$

Na forma canônica, a solução básica para o problema é representada pelas equações 2.14, 2.15 e 2.16:

$$f = \overline{f} \qquad (2.14)$$

$$\overline{x}_m = \overline{b}_m \qquad (2.15)$$

$$x_{m+1} = x_{m+2} = x_n = 0 \qquad (2.16)$$

Para garantir que a solução básica seja viável, é necessário que todos os valores de $\overline{b}_i$ sejam positivos. Quando isso não ocorre, deve-se buscar outro ponto, com diferentes variáveis básicas. Finalmente, quando $\overline{b}_i$ é maior ou igual a 0, dizemos que ele está em sua forma canônica viável (EDGAR, HIMMELBLAU e LASDON, 2001). Nesta forma, as variáveis básicas apresentam valor igual a $\overline{b}_{i,}$ e as demais não-básicas são nulas.

Conforme demonstrado por Bazaraa, Jarvis e Sherali (2010), se existe uma solução ótima para a função, esta solução corresponderá a um ponto extremo ótimo. Ao encontrar uma solução básica viável, estamos encontrando um ponto extremo da função (ponto no limite da região viável), que nos dá um ponto de partida para aplicação da segunda etapa do método Simplex. No entanto, antes de

proceder, ainda é possível fazer uma análise mais profunda da solução básica encontrada, observando sua forma canônica, para determinar se ela é mesmo ótima, ou se há uma alternativa mais adequada.

Podemos reescrever a função objetivo do sistema como a equação 2.17:

$$f = \overline{f} + \overline{c}_{m+1}x_{m+1} + \cdots + \overline{c}_n x_n \tag{2.17}$$

Uma solução básica viável será uma solução mínima viável se todas as constantes $\overline{c}_{m+1}, \overline{c}_{m+2}, \ldots, \overline{c}_n$ (chamadas de custos reduzidos) forem não-negativas (EDGAR, HIMMELBLAU e LASDON, 2001).

Uma vez que as variáveis x_{m+1}, …, x_n são nulas na solução básica (e não podem ser negativas), dado que todos os custos reduzidos são positivos ou nulos, o aumento no valor de qualquer uma delas levaria a um aumento do valor da função que se quer minimizar, o que pode ser entendido como um distanciamento do ponto ótimo.

No caso em que um dos custos reduzidos $\overline{c}_k$ seja nulo, a variação de x_k não altera o valor da função objetivo, e existem múltiplos ótimos para diferentes valores de x_k. Desta forma, observamos que para que uma solução básica viável seja mínima e única, é necessário que os custos reduzidos $\overline{c}_n$ sejam todos positivos não-nulos.

Quando ao menos um dos custos reduzidos tem valor negativo, o aumento no valor da variável correspondente implica diminuição no valor da função objetivo, e isto significa que a presente solução básica não é ótima, havendo possibilidade de aprimoramento. Para tal, faz-se o pivoteamento em torno da variável cujo coeficiente $\overline{c}_n$ na função objetivo é negativo. O maior valor válido desta variável, considerando as restrições de positividade para as outras variáveis básicas, corresponderá à solução mínima, e ela substituirá uma das variáveis básicas, cujo valor passará a ser nulo. Este processo pode ser iterado até que todos os custos reduzidos sejam positivos.

Para ilustrar, analisemos um exemplo adaptado de EDGAR, HIMMELBLAU e LASDON (2001). O sistema composto por 2.18, 2.19 e 2.20 tem uma solução básica viável em 2.21.

$$x_1 - 0{,}75\, x_2 + 2\, x_3 - 0{,}25\, x_4 = 3 \quad (2.18)$$
$$x_5 - 0{,}25\, x_2 + 3\, x_3 - 0{,}75\, x_4 = 5 \quad (2.19)$$
$$-f + 8\, x_2 - 24\, x_3 + 5\, x_4 = -28 \quad (2.20)$$

$$x_1 = 3, x_2 = x_3 = x_4 = 0, x_5 = 5, f = 28 \quad (2.21)$$

O custo reduzido $\overline{c}_3$ = -24 é negativo, portanto o pivoteamento deve ser feito em torno da variável x_3. O maior valor que x_3 pode assumir deve satisfazer as restrições 2.22, nas quais as variáveis não-básicas são nulas, para que as variáveis básicas x_1 e x_5 continuem positivas.

$$x_1 = 3 - 2\, x_3$$
$$x_5 = 5 - 3\, x_3 \quad (2.22)$$

Neste caso, o maior valor possível para x_3 é 1,5, para o qual x_1 assume valor nulo (passando a ser variável não-básica), x_5 é 0,5 e a função objetivo tem valor -8. O pivoteamento é então, feito na equação 2.18, que contém a variável básica x_1, e retorna o sistema 2.23, 2.24 e 2.25. Este sistema tem a solução básica viável 2.26.

$$x_3 + 0{,}5\, x_1 - 0{,}375\, x_2 - 0{,}125\, x_4 = 1{,}5 \quad (2.23)$$
$$x_5 - 1{,}5\, x_1 + 0{,}875\, x_2 - 0{,}375\, x_4 = 0{,}5 \quad (2.24)$$
$$-f + 12\, x_1 - x_2 + 2\, x_4 = 8 \quad (2.25)$$

$$x_1 = x_2 = 0, x_3 = 1{,}5, x_4 = 0, x_5 = 0{,}5, f = -8 \quad (2.26)$$

A solução (2.26) está mais próxima da otimalidade do que a solução anterior (2.11), pois assume um valor menor, mas ainda assim pode ser aprimorada, uma

vez que o custo reduzido associado à variável x_2 é negativo. Sendo assim, a próxima etapa de pivoteamento deve ser feita em torno dela.

3 – METODOLOGIA

Os problemas de programação linear abordados neste trabalho foram modelados utilizando Pyomo, uma linguagem de modelagem algébrica (*algebraic modeling language*; ou AML) *open source* para Python que permite a formulação de modelos matemáticos para aplicação de otimização.

O desenvolvimento de um modelo consiste na representação simplificada de um problema real. No Pyomo, um modelo matemático pode ser construído de duas formas: abstrata ou concreta, que correspondem a classes distintas de objetos do Python. A principal diferença entre as duas está na forma com que os parâmetros (dados) são incorporados ao problema, sendo que no primeiro caso, eles não estão determinados *a priori*, e a especificação de dados cria uma instância do modelo. Em contrapartida, os parâmetros são fornecidos já durante a definição de um modelo concreto.

Parâmetros são declarados como instâncias da classe *Param*, e podem ser indexados por objetos da classe *Se*t. As variáveis do problema, por sua vez, são declaradas com a função *Var*, através da qual podemos especificar seus domínios e limites superior e inferior, e podem ser indexadas da mesma forma que os parâmetros.

Usualmente, quando modelamos tanto as restrições quanto a função objetivo no Pyomo, estas são declaradas em função de outras funções, definidas como a regra (*rule*) da função *Objective* ou *Constraint*. Essas outras funções, por sua vez, são declaradas como instâncias da classe *Expression*.

Para ilustrar, foram extraídos dois modelos simplificados da documentação do Pyomo (HART, WILLIAM E *et al.*, 2017), que podem ser encontrados no apêndice A.

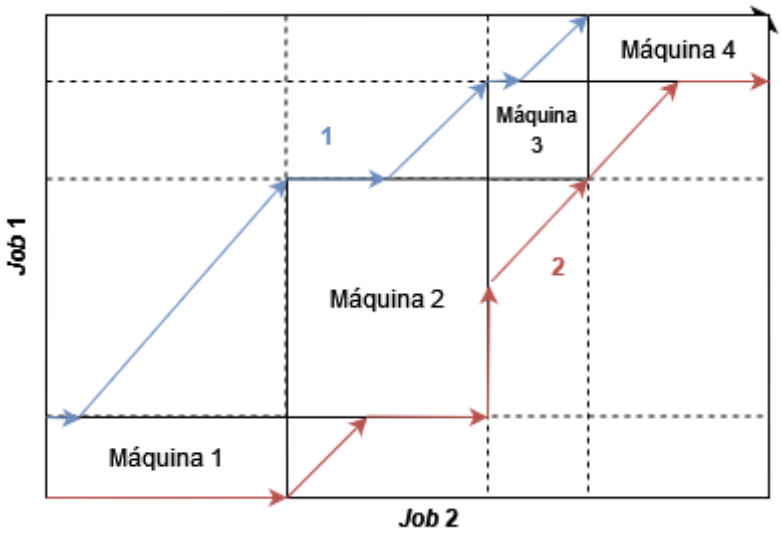
O Pyomo suporta a resolução de problemas de otimização usando múltiplos *solvers* distintos, e inclusive conta com interfaces especializadas para alguns deles. Por padrão, o solver utilizado é o GLPK (*GNU Linear Programming Kit*). Neste

trabalho, foram utilizados o *COIN-OR Branch-and-cut* e o *COIN-OR Interior Point Optimizer (IPOPT)*, cada um em um estudo de caso (ambos modelos concretos), visando a elaboração de material auxiliar para o estudo da disciplina de Síntese e Otimização de Processos Químicos.

3.1 – Problema de *flow shop*

O primeiro estudo de caso, extraído de VACHAJITPAN (1981), é um exemplo do clássico problema de *flow shop* estático, no qual todos os *produtos* estão disponíveis para processamento no tempo zero. Cada um destes *produtos* deve passar por quatro operações, sendo cada uma destas realizada em uma máquina distinta, e a ordem das operações não é comutativa. O processamento em cada máquina deve ser ininterrupto, e uma mesma máquina não pode processar dois *produtos* simultaneamente, mas uma máquina pode operar processando um *produto i*, enquanto outra processa um produto *j*. A figura 3.1 ilustra um exemplo de problema com 2 *produtos* e 4 máquinas. Duas possíveis sequências de operação que atendem estas condições estão representadas pelas linhas 1 e 2, destacadas na figura.

Figura 3.1. Exemplo de problema *flow shop*



Fonte: Elaboração própria

Um problema *flow shop* de *i produtos* e *k* operações, cada uma conduzida em uma máquina, consiste na minimização do tempo total de processo fabril, isto é, todo o tempo decorrido a partir do início da primeira operação até o final da última. Para tal, o tempo de cada operação é otimizado através de sequenciamento eficiente dos *produtos* em cada máquina.

Seja P_{ik} a duração de operação do *produto i* na máquina *k* (constante conhecida); T_{ik}, uma variável que representa o momento de início do *produto i* na máquina *k*; S_k, o momento em que a máquina *k* é iniciada; e Y_{jik}, uma variável booleana que estabelece a relação de precedência dos *produtos* nas máquinas, assumindo valor 1 quando o *produto j* sucede o *i* na máquina *k*, e caso contrário, valor nulo.

Temos uma restrição de desigualdade (equação 3.1) e uma de igualdade (3.2), respectivamente:

$$T_{i,k+1} \geq T_{ik} + P_{ik} \tag{3.1}$$

$$T_{ik} = S_k + \sum_{J \neq I} P_{jk}\, Y_{jik} \tag{3.2}$$

A equação (3.1) fornece o horário de início da operação *k + 1*, que logicamente deve ser após o tempo de início da operação anterior acrescido do tempo decorrido durante a operação.

Em (3.2), o tempo de início da operação *k* é a soma dos tempos decorridos de todas as operações *k* dos *produtos* que vieram antes do *produto i* ($Y_{jik} = 1$), desde a ativação da máquina *k*.

Deste modo, o problema genérico fica composto de *mn* variáveis T_{ik}, *m* variáveis S_k, *mn(n – 1)* variáveis Y_{jik}, *(m – 1)n* restrições de desigualdade e *mn* restrições de igualdade. Para um caso como o do artigo, com 3 *produtos* e 4 máquinas, temos um total de 28 variáveis e 21 restrições. A função objetivo a ser otimizada não foi explicitada.

No artigo, o problema foi resolvido pelo método Simplex. No Pyomo, dois modelos distintos foram construídos.

3.2 – Problema de alocação de água

O segundo caso estudado foi apresentado em 1979 por N. TAKAMA, T. KURIYAMA, K. SHIROKO e T. UMEDA, sob o título de *Optimal water allocation in a petroleum refinery*. O problema consiste na minimização dos custos do investimento para implementação de um sistema de tratamento de água de efluente em uma refinaria de petróleo, em um horizonte de tempo de sete anos e meio.

O sistema de tratamento foi integrado a um sistema maior, que consiste de oito subsistemas *i*, sendo o primeiro (*i* = 1) a fonte de água limpa, três subsistemas que utilizam água (*i* = 2, 3, 4), os três de tratamento a serem implementados (*i* = 5, 6, 7) e um último de despejo (*i* = 8).

A água utilizada pelos subsistemas *i* = 2, 3, 4 segue para os subsistemas de tratamento, podendo retornar ou não a eles. O que define o trajeto que a água percorre ao sair de um subsistema *j* em direção a outro subsistema *i* são as *razões de separação* representadas pelas variáveis δ_{ij}. As vazões de efluente de cada subsistema *i* foram modeladas como as variáveis Q_i, e tem valores pré-determinados para os subsistemas que utilizam água, que foram implementados como restrições, sendo Q_2 = 45,8 ton/h, Q_3 = 32,7 ton/h e Q_4 = 56,5 ton/h.

Ao longo do processo de refinamento de petróleo, nos três subsistemas que utilizam água, são produzidos três poluentes *k*: sulfeto de hidrogênio, óleo e partículas sólidas suspensas. Cada subsistema tem uma taxa característica de geração de poluente (modelados no Pyomo como o *Param P*), da mesma forma que cada subsistema de tratamento tem uma de remoção (*Param r*). Esses parâmetros estão listados nas tabelas 3.1 e 3.2.

Tabela 3.1. Taxa de geração de poluentes (P_{ik}) nos subsistemas que utilizam água, em ton/h

Subsistema *i*	Poluente 1 (H_2S)	Poluente 2 (óleo)	Poluente 3 (part.)
2	0,0179	0,0005	0,0012
3	0,536	0,0033	0,0005
4	0,0013	0,0057	0,002
demais	0	0	0

Fonte: Adaptado de N. Takama *et al.*, 1979.

Tabela 3.2. Taxa de remoção de poluentes (r_{ik}) nos subsistemas de tratamento de água, em porcentagem

Subsistema *i*	Poluente 1 (H_2S)	Poluente 2 (óleo)	Poluente 3 (part.)
5	99,9	0	0
6	0	95	20
7	90	90	97
demais	0	0	0

Fonte: Adaptado de N. Takama *et al.*, 1979.

Os balanços de massa para a água e para os três poluentes em cada subsistema, representados respectivamente pelas equações (3.3) e (3.4), constituem as restrições de igualdade do problema. Há ainda uma série de restrições de desigualdade, sumarizadas pela equação (3.5), que determinam a quantidade máxima de poluente que pode estar presente na corrente de alimentação de cada subsistema *i* (parâmetro z_i, cujos valores podem ser encontrados na tabela 3.3). É válido para $i = 2, ..., n$:

$$Q_i = \sum_{j \neq i}^{n} \delta_{ij}\, Q_j \tag{3.3}$$

$$Q_i \cdot x_i^k = \left(1 - r_i^k\right) \sum_{j \neq i}^{n} (\delta_{ij} \cdot Q_j \cdot x_j^k) + P_i^k \tag{3.4}$$

$$\frac{\sum_{j \neq i}^{n} (\delta_{ij} \cdot Q_j \cdot x_j^k)}{Q_i} \leq z_i^k \tag{3.5}$$

Tabela 3.3. Limite de poluente *k* na alimentação de *i*, em ppm

Subsistema *i*	Poluente 1 (H_2S)	Poluente 2 (óleo)	Poluente 3 (part.)
2	0	0	0
3	500	20	50
4	20	120	50
8	2	2	5
demais	-	-	-

Fonte: Adaptado de N. Takama *et al.*, 1979.

Foram inclusas também duas restrições adicionais, que não estavam declaradas no artigo original. A equação (3.6) estabelece que para cada subsistema *j,* a soma das *razões de separação* da corrente de efluente não pode exceder 1, ou seja, reforça o balanço de massa. Já a equação (3.7) representa o balanço de massa global do sistema.

$$\sum_{j \neq i}^{n} \delta_{ij} = 1 \tag{3.6}$$

$$Q_1 = Q_8 \tag{3.7}$$

No artigo, o algoritmo de otimização resultou em um sistema sem reciclos, onde a água segue sempre de um subsistema *j* para um subsistema *i*, sendo *i* > *j*.

Esta particularidade foi incorporada aos balanços de massa do modelo desenvolvido no Pyomo. Quando ela não é considerada, o sistema resultante é composto de dezenas de correntes de reciclo. Este caso não foi abordado no trabalho.

Por fim, a função objetivo equivale à soma dos custos de investimento divididos ao longo dos sete anos e meio previstos para *payback*, mais os custos anuais de operação e da alimentação de água utilizada. A planta operará durante 8 mil horas em um ano, e o custo de água por tonelada é de $ 0,3. Os demais parâmetros para as equações (3.8) a (3.11) estão dispostos na tabela 3.4.

Tabela 3.4. Custos de investimento e operação dos sistemas de tratamento de água

Subsistema *i*	**Custo de investimento ($)**	**Custo de operação ($/h)**
5	16800 x $Q^{0,7}$	Q
6	4800 x $Q^{0,7}$	0
7	12600 x $Q^{0,7}$	0,0067 x Q

Fonte: Adaptado de N. Takama, T. Kuriyama, K. Shiroko e T. Umeda, 1979.

$$custo\ de\ operação = 8000 \cdot (Q_5 + 0{,}0067 \cdot Q_7) \quad (3.8)$$

$$custo\ de\ investimento = 16800 \cdot {Q_5}^{0,7} + 4800 \cdot {Q_6}^{0,7} + 12600 \cdot Q_7^{0,7} \quad (3.9)$$

$$custo\ de\ água = 8000 \cdot 0{,}3 \cdot Q_1 \quad (3.10)$$

$$custo\ total = custo\ de\ operação + \frac{custo\ de\ investimento}{7{,}5} + custo\ de\ água \quad (3.11)$$

O problema do artigo não é linear, e apresenta rigorosas restrições de desigualdade. Devido a estas características, os autores optaram por utilizar um método complexo de resolução, em detrimento de métodos de programação. Este método introduz funções de penalidade para transformar o problema original em

uma série de problemas sem restrições de desigualdade. Em contrapartida, neste trabalho, o problema foi modelado fielmente à sua forma original.

4 – RESULTADOS E DISCUSSÃO

4.1 – Problema de *flow shop*

O algoritmo convergiu para ambas as modelagens, chegando a soluções distintas do artigo. Os tempos de início de cada máquina obtidos na modelagem do artigo, e nas desenvolvidas para este trabalho, são apresentados na tabela 4.1.

Tabela 4.1 Comparativo entre os tempos de início e de duração de cada operação em cada máquina, em diferentes modelagens

Máquina	***Produto***	P_{ik}	T_{ik} **(artigo)**	T_{ik} **(modelo 1.1)**	T_{ik} **(modelo 1.2)**	T_{ik} **(modelo 1.3)**	T_{ik} **(modelo 2)**
1	1	6	1	0	1	5	5
	2	1	0	0	0	0	0
	3	4	7	0	7	1	1
2	1	8	7	6	7	11	11
	2	3	4	1	4	3	5
	3	5	15	9	15	6	3
3	1	9	16	14	16	20	19
	2	9	7	5	7	11	10
4	1	4	28	23	25	29	28
	2	6	16	21	23	23	22
	3	6	22	17	25	17	16

Fonte: Elaboração própria.

4.1.1 – Modelo 1

Nesta subseção, abordaremos as três iterações do primeiro modelo. Na primeira, foi utilizada uma construção fiel à modelagem do artigo como MILP, declarando P_{ik} como Param e T_{ik}, S_k e Y_{jik} como Var inteiras. As $(m - 1)n$ restrições

de desigualdade determinam as relações de precedência entre cada operação, e as *mn* restrições de igualdade estabelecem o cronograma. Foi ainda criada uma variável Z a ser minimizada como função objetivo, que representa o instante final do processamento, e está sujeita a uma restrição (equação 4.1) que se assemelha às restrições de igualdade:

$$Z \geq T_{ik} + P_{ik} \tag{4.1}$$

Um limite superior, cujo valor corresponde a 61, a soma dos parâmetros P_{ik}, foi imposto às variáveis T_{ik} e Z. O *solver* utilizado foi o Cbc (*COIN-OR Branch-and-Cut*).

Observando os resultados obtidos, expostos na tabela 4.2, foi constatado que apenas as variáveis Y_{jik} não eram capazes de assegurar as relações de precedência na primeira máquina (k = 1), uma vez que todas Y_{ji1} assumiram valor nulo. Como consequência disto, todos os *produtos* iniciaram simultaneamente, tal qual visto anteriormente na tabela 4.1 e demonstrado na figura 4.1.

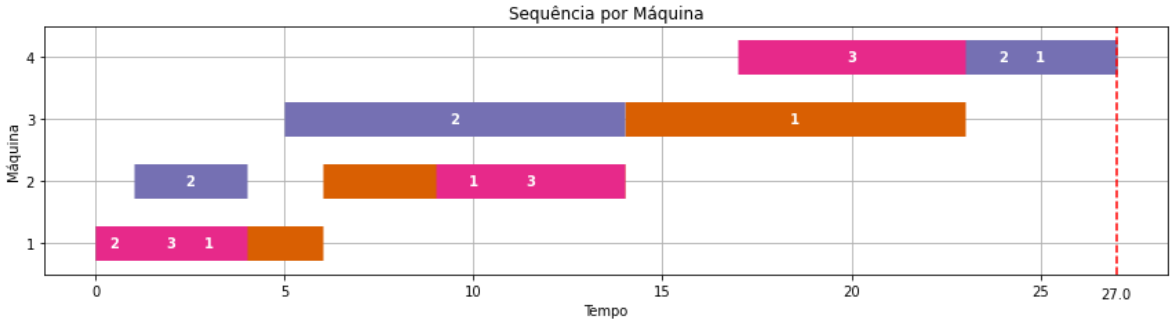
Tabela 4.2. Comparativo entre os valores assumidos pelas variáveis inteiras booleanas, Y_{jik}, em cada modelagem

Variável inteira	Artigo	1ª iteração	2ª iteração	3ª iteração
Y_{121}	0	0	0	0
Y_{131}	1	0	1	0
Y_{211}	-	0	1	1
Y_{231}	1	0	1	1
Y_{311}	-	0	0	1
Y_{321}	-	0	0	0
Y_{122}	0	0	0	0
Y_{132}	1	1	1	0

Y_{212}	-	0	1	1
Y_{232}	1	0	1	1
Y_{312}	-	1	0	1
Y_{322}	-	0	0	0
Y_{123}	0	0	0	0
Y_{133}	1	1	1	0
Y_{213}	-	1	1	1
Y_{233}	0	0	1	0
Y_{313}	-	-	0	1
Y_{323}	-	-	0	1
Y_{124}	0	1	1	0
Y_{134}	0	0	0	0
Y_{214}	-	0	0	1
Y_{234}	1	0	1	0
Y_{314}	-	1	1	1
Y_{324}	-	0	0	1

Fonte: Elaboração própria.

Figura 4.1. Gráfico de Gantt da sequência de operações em cada máquina para a primeira iteração do modelo 1



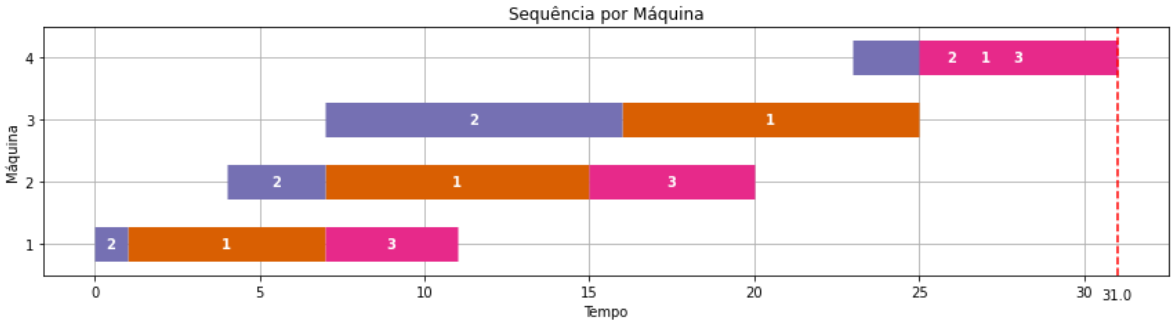
Fonte: Elaboração própria.

Para endereçar estes problemas, foi introduzida uma restrição adicional, representada pela equação 4.2, que caracteriza a segunda iteração do modelo. Se Y_{jik} é 1, e portanto, o *produto j* precede *i* na máquina *k*, então logicamente Y_{ijk} deve ter valor nulo, pois as duas situações são mutuamente excludentes.

$$Y_{ijk} + Y_{jik} = 1 \tag{4.2}$$

Após a introdução desta restrição, as três primeiras máquinas apresentaram resultados idênticos ao artigo. No entanto, a quarta máquina ficou presa em um *loop* teórico, no qual $Y_{124} = Y_{234} = Y_{314} = 1$. Como podemos observar na figura Y, isto fez com que parte do processamento de todos os *produtos* ocorresse simultaneamente.

Figura 4.2. Gráfico de Gantt da sequência de operações em cada máquina para a segunda iteração do modelo 1



Fonte: Elaboração própria.

Para quebrar este *loop*, e prevenir a ocorrência de outro possível *loop*, $Y_{214} = Y_{134} = Y_{324} = 1$, foram introduzidas mais duas restrições, representadas pelas equações 4.3 e 4.4, que se aplicam a todas as máquinas:

$$Y_{12k} + Y_{23k} + Y_{31k} \leq 2 \tag{4.3}$$

$$Y_{21k} + Y_{13k} + Y_{32k} \leq 2 \quad (4.4)$$

Isso nos leva à terceira e última iteração. O principal aspecto que difere a solução dos dois modelos da do artigo é a ordem de processamento dos *produtos* em cada máquina:

- No artigo, o primeiro *produto* processado na máquina 1 é o 2, seguido do *produto* 1, que precede o 3. No modelo 1, a ordem dos dois últimos se inverte. O mesmo ocorre na máquina 2.
- Devido a esta inversão, como o *produto* 1 é o último na máquina 2, embora a operação da máquina 3 se inicie com o *produto* 2, para que ela seja ininterrupta, é necessário que ela inicie mais tarde, no minuto 11, em vez do 7, como no artigo. Isto não é um fator que influencia, por si só, no tempo final de todo o processo.
- A sequência na máquina 4 é completamente diferente: 2, 3 e 1 no artigo, e 3, 2 e 1 no modelo 1.

Isso nos mostra que o problema em questão possui mais de uma solução ótima. Além disso, no caso específico do modelo 1, o processamento do *produto* 1 na máquina 3 se inicia um minuto após o término na máquina 2, resultando num tempo final de 33 minutos. Considerando as restrições do modelo, não há uma explicação aparente para este desvio da otimalidade.

4.1.2 – Modelo 2

O segundo modelo adota uma abordagem um pouco diferente do problema. O parâmetro P_{ik} e as variáveis T_{ik} e Z são declarados da mesma forma que no modelo 1, e têm o mesmo limite superior. A função objetivo, as restrições de igualdade e as de desigualdade associadas a Z também são idênticas.

A primeira diferença fundamental entre os dois modelos é a indexação das variáveis e restrições utilizando *Sets* construídos com base em um dicionário, que contém a duração (*P*) e a relação de precedência de cada operação em cada máquina.

Essa relação de precedência é utilizada para restringir o início de cada operação (*j, l*), baseado no término da operação (*i, k*) que a precede, conforme equação 4.5.

$$T_{jl} + P_{jl} \leq T_{ik} \tag{4.5}$$

Neste contexto, *i* = *j*, mas esta abordagem permite a modelagem de requerimentos mais complexos e interdependência de *produtos*.

O segundo aspecto que diferencia um modelo do outro é a inclusão de disjunções, conforme equação 4.6, que asseguram que duas operações não podem ser conduzidas simultaneamente em uma mesma máquina, papel que é desempenhado por restrições no modelo 1.

$$\left[\, T_{ik} + P_{ik} \leq T_{jk} \,\right] \bigvee \left[\, T_{jk} + P_{jk} \leq T_{ik} \,\right] \tag{4.6}$$

Para garantir o funcionamento ininterrupto das máquinas, foram ainda criadas mais duas variáveis auxiliares, *minmach* e *maxmach*, que assumem os valores de tempo inicial e final de funcionamento de cada máquina quando submetidas às restrições representadas pelas equações 4.7, 4.8 e 4.9:

$$T_{ik} \geq minmach_k \tag{4.7}$$

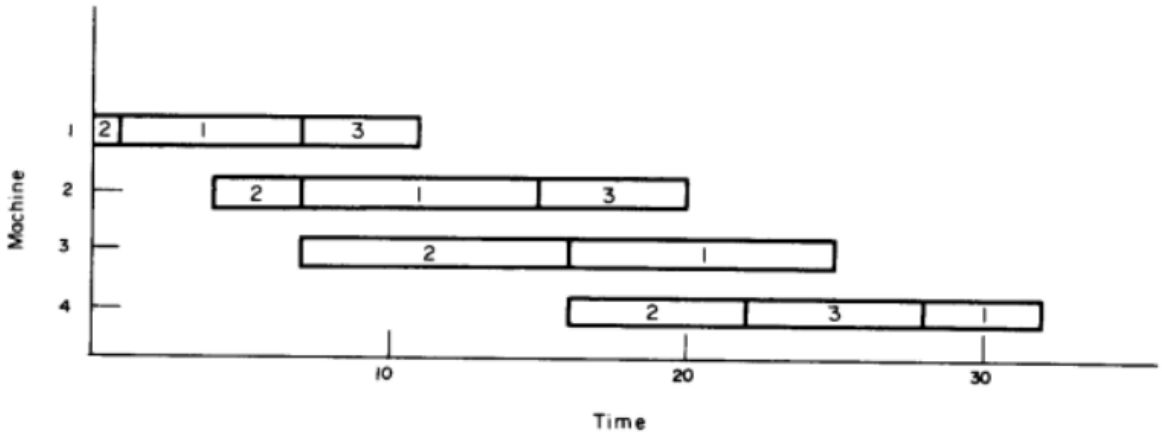
$$T_{ik} + P_{ik} \leq maxmach_k \tag{4.8}$$

$$\sum P_{ik} = maxmach_k - minmach_k \tag{4.9}$$

O modelo 2 não enfrentou problemas, e chegou a uma sequência final de operações idêntica à da terceira iteração do primeiro modelo. O término da última operação foi a marca de 32 minutos, tal como no artigo original.

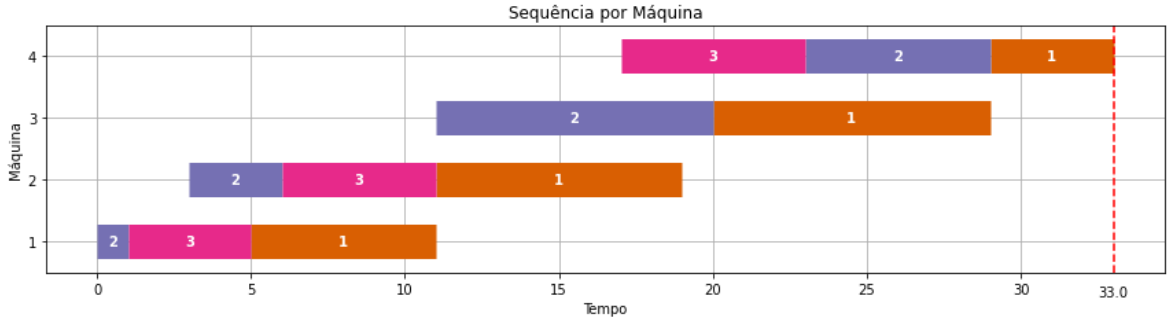
As figuras 4.3, 4.4 e 4.5 estabelecem um comparativo entre os gráficos de Gantt das versões finais dos dois modelos desenvolvidos e do artigo.

Figura 4.3. Gráfico de Gantt da sequência de operações em cada máquina para o modelo do artigo



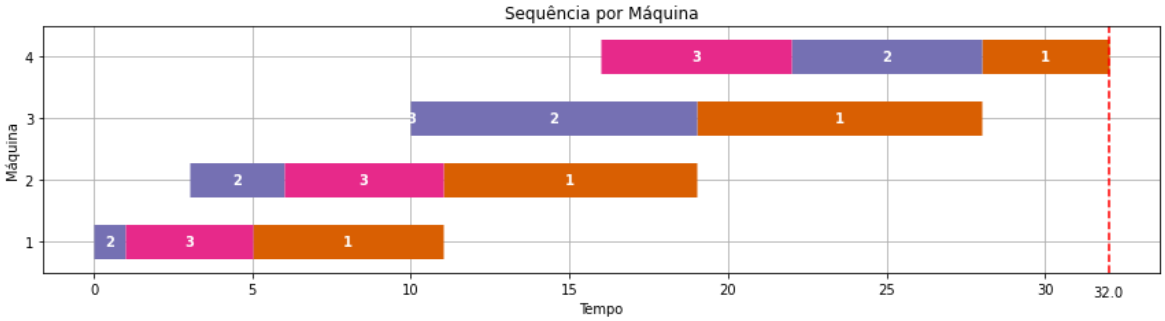
Fonte: Vachajitpan, 1981.

Figura 4.4. Gráfico de Gantt da sequência de operações em cada máquina para a iteração final do modelo 1



Fonte: Elaboração própria.

Figura 4.5. Gráfico de Gantt da sequência de operações em cada máquina para o modelo 2



Fonte: Elaboração própria.

4.2 – Problema de alocação de água

Os dois conjuntos de resultados para as vazões de cada subsistema estão expostos na tabela 4.3. A tabela 4.4 contém uma comparação dos valores das *razões de separação* dos modelos do artigo e do Pyomo. Variáveis com índices omitidos são nulas.

Tabela 4.3. Vazão de efluente (Q_i) do subsistema *i*

Vazão (ton/h)	Artigo	Pyomo
Q_1	102,3	101,81
Q_2	45,8	45,8
Q_3	32,7	32,7
Q_4	56,5	56,5
Q_5	45,8	45,31
Q_6	87,3	82,05
Q_7	102,3	101,81
Q_8	102,3	101,81

Fonte: Elaboração própria.

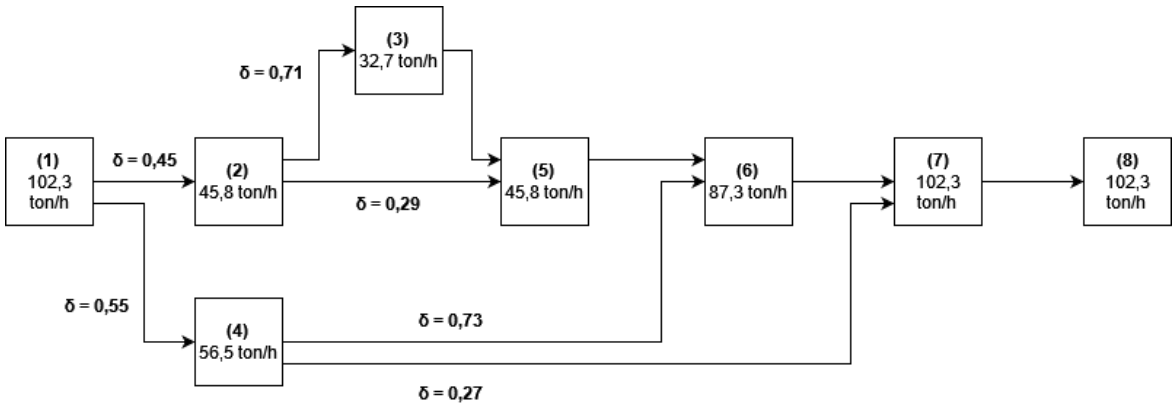
Tabela 4.4. Comparativo dos valores das *razões de separação* para os modelos do artigo e do Pyomo

Razão de separação	**Artigo**	**Pyomo**
δ_{21}	0,45	0,45
δ_{32}	0,71	0,71
δ_{41}	0,55	0,55
δ_{42}	0	0,01
δ_{52}	0,29	0,28
δ_{53}	1	1
δ_{64}	0,73	1
δ_{65}	1	0,56
δ_{74}	0,27	0
δ_{75}	0	0,44
δ_{76}	1	1
δ_{87}	1	1

Fonte: Elaboração própria.

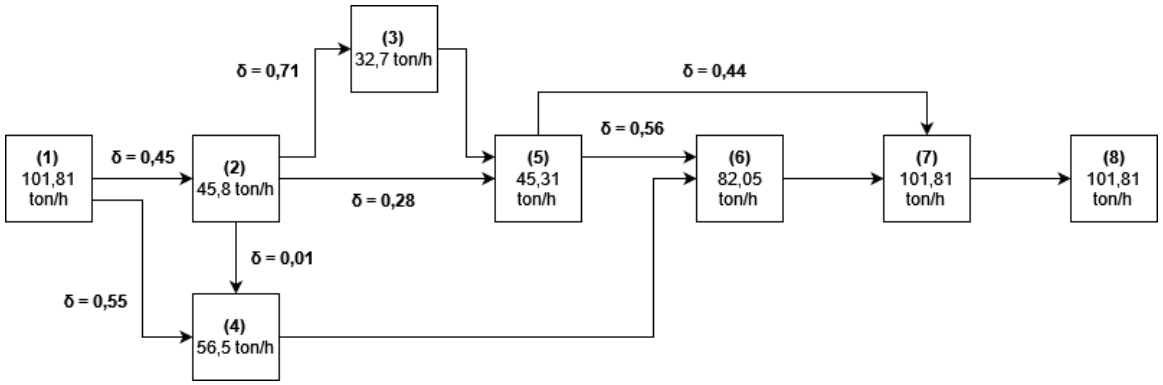
Os esquemas das figuras 4.6, 4.7 e 4.8 representam o caminho que a água percorre nos três sistemas, e ilustram o efeito das divergências nas *razões de separação*.

Figura 4.6. Fluxograma simplificado do sistema do artigo



Fonte: Adaptado de N. Takama *et al.*, 1979.

Figura 4.7. Fluxograma simplificado do sistema do Pyomo



Fonte: Elaboração própria.

A tabela 4.5 mostra um comparativo entre as quantidades de poluente presentes na alimentação de cada subsistema *i*, calculadas conforme balanço de massa descrito na equação (3.5).

Tabela 4.5. Quantidade de poluentes na alimentação de cada subsistema, em ppm

	Artigo			Pyomo		
Subsistema *i*	**k = 1**	**k = 2**	**k = 3**	**k = 1**	**k = 2**	**k = 3**
2	0	0	0	0	0	0
3	390	10	25	390,82	10,91	26,2
4	0	0	0	3,17	0,09	0,21
8	2	1.9	0.9	2	2	0,91

Fonte: Elaboração própria.

Observa-se que todos os subsistemas obedecem às restrições de quantidade de cada poluente. Vale ressaltar que, no modelo do artigo, toda a alimentação do subsistema 4 é proveniente da fonte de água (*i* = 1); no modelo do Pyomo, no entanto, 1% do efluente do subsistema 2 (o que corresponde a

aproximadamente 458 kg/h) e 55% do efluente de 1 (aproximadamente 56 ton/h) são destinados ao subsistema 4.

Na tabela 4.6, estão expostos os custos de água, de operação e de investimento dos dois modelos. Podemos constatar que, com uma pequena margem de erro, o algoritmo do Pyomo, usando o solver *IPOPT,* foi capaz de chegar a resultados melhores que os do artigo. Observa-se que o balanço de massa para o subsistema 1 tem uma pequena divergência, e podemos inferir que isso se deve à uma aproximação feita pelo *solver* para garantir a convergência. O método complexo do artigo opera através da resolução de sucessivos problemas lineares, e não podemos descartar a hipótese de que isso leve a alguma discrepância nos resultados, que é refletida na solução final.

Tabela 4.6. Custos de implantação e operação do sistema de tratamento de água

Custo	Unidade	Artigo	Pyomo
Função objetivo	10^3 $/ano	707	701
Investimento	10^3 $	676	668
Operação	10^3 $/ano	371	368
Água	10^3 $/ano	246	244

Fonte: Elaboração própria.

5 – CONCLUSÕES E SUGESTÕES

A resolução dos dois problemas de otimização com o auxílio dos algoritmos escritos no Pyomo produziu resultados muito similares aos obtidos pelos modelos originais, empregados nos artigos que apresentaram estes problemas, que fizeram uso de algoritmos tradicionais. Pequenas variações no desempenho e resultados foram encontradas ao longo do desenvolvimento de cada modelo, e elas podem ser atribuídas às estratégias de modelagem e ao próprio funcionamento distinto dos *solvers* disponíveis no Pyomo.

Além disso, é válido ressaltar que nenhum dos dois casos estudados pode ser modelado de forma estritamente igual à exposta no artigo; foi preciso adicionar algumas restrições e variáveis incrementais a cada modelo. Um exemplo notório de informação omitida foi a função objetivo do problema de *flow shop*, que pode ser construída de mais de uma forma, e não se pode descartar a possibilidade de que isto influencie os resultados finais. No caso do problema de alocação da água, não é mencionado no artigo se foi estabelecida uma restrição ao emprego de correntes de reciclo no sistema; o resultado final é apresentado sem reciclos, mas a não-inclusão desta restrição altera completamente os resultados obtidos quando se aplica o algoritmo construído no Pyomo. Tudo isto coloca em evidência a importância de adaptar o problema que se deseja resolver ao *solver* utilizado.

Algumas sugestões para trabalhos futuros incluem a generalização do problema de *flow shop* para *n produtos* e *m* máquinas, utilizando modelo similar ao modelo 1, que necessitaria o desenvolvimento de uma lógica robusta para impedir a formação de *loops* inconsistentes; e a análise de um sistema alternativo de alocação de água que inclua reciclos, bem como a comparação da performance de outros *solvers* quando aplicados a cada um dos problemas.

REFERÊNCIAS BIBLIOGRÁFICAS

BAZARAA, Mokhtar S.; JARVIS, John J.; SHERALI, Hanif D.. Linear Programming and Network Flows. 4. ed. Hoboken, Nova Jersey: John Wiley & Sons, Inc., 2010.

DANTZIG, George; THAPA, Mukund. Linear Programming 1: Introduction. Nova Iorque: Springer-Verlag New York Inc., 1997.

EDGAR, Thomas; HIMMELBLAU, David; LASDON, Leon. Optimization of Chemical Processes**.** 2. ed. Nova Iorque: McGraw-Hill Companies Inc., 2001.

HART, William E., Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. Pyomo – Optimization Modeling in Python. Second Edition. Vol. 67. Springer, 2017.

HART, William E., Jean-Paul Watson, and David L. Woodruff. "Pyomo: modeling and solving mathematical programs in Python." Mathematical Programming Computation 3, 2011.

KANTOR, Jeffrey. *ND Pyomo Cookbook.* GitHub. <https://github.com/jckantor/ND-Pyomo-Cookbook>, 2018

TAKAMA, N.; KURIYAMA, T.; SHIROKO, K.; UMEDA, T.. Optimal water allocation in a petroleum refinery. Computers & Chemical Engineering, [S.L.], v. 4, n. 4, p. 251-258. Elsevier BV, 1980.

VACHAJITPAN, Porpan. Job sequencing with continuous machine operation. Computers & Industrial Engineering, [S.L.], v. 6, n. 3, p. 255-259. Elsevier BV, 1982.

APÊNDICE A – Exemplos de modelo concreto e abstrato

Seja o problema:

$$\min 2x_1 + 3x_2$$
$$sujeito\ a:$$
$$3x_1 + 4x_2 \geq 1$$
$$x_1, x_2 \geq 0$$

Ele pode ser modelado de forma concreta da seguinte maneira:

```
model = pyo.ConcreteModel()
model.x = pyo.Var([1,2], domain=pyo.NonNegativeReals)
model.OBJ = pyo.Objective(expr=2*model.x[1] + 3*model.x[2])
model.Constraint1 = pyo.Constraint(expr = 3 * model.x[1] + 4 * m
odel.x[2] >= 1)
```

Observe que aqui, a *Var* x é indexada por um *Set* que não foi declarado, por conter apenas dois elementos. Outra maneira de se declarar esta variável seria:

```
model.i = pyo.Set(initialize=[1,2])
model.x = pyo.Var(model.i, domain=pyo.NonNegativeReals)
```

Para um problema com estrutura similar ao anterior, mas generalizada para *j* variáveis e *i* restrições de desigualdade:

$$\min \sum_{j=1}^{n} c_j\, x_j$$
$$sujeito\ a:$$
$$\sum_{j=1}^{n} a_{ij}\, x_j \geq b_i$$
$$x_j \geq 0$$

$$i = 1, \ldots, m$$

$$j = 1, \ldots, n$$

Temos um exemplo de modelo abstrato:

```
model = pyo.AbstractModel()

model.m = pyo.Param(within=pyo.NonNegativeIntegers)
model.n = pyo.Param(within=pyo.NonNegativeIntegers)

model.I = pyo.RangeSet(1, model.m)
model.J = pyo.RangeSet(1, model.n)

model.a = pyo.Param(model.I, model.J)
model.b = pyo.Param(model.I)
model.c = pyo.Param(model.J)

model.x = pyo.Var(model.J, domain=pyo.NonNegativeReals)

def obj_expression(m):
    return pyo.summation(m.c, m.x)
model.OBJ = pyo.Objective(rule=obj_expression)

def ax_constraint_rule(m, i):
    return sum(m.a[i,j] * m.x[j] for j in m.J) >= m.b[i]
model.AxbConstraint = pyo.Constraint(model.I,  rule=\ ax_constra
int_rule)
```

Nota-se que a forma de se declarar os *Param* difere um pouco do modelo concreto, e os *Sets* são criados dependentes de um intervalo, que será especificado depois. As restrições são geradas a partir de uma regra que é avaliada para cada elemento de *i*, resultando em *i* equações.

APÊNDICE B – Algoritmo para resolução do problema de *flow shop*

Este apêndice contém o código em sua versão final do primeiro modelo desenvolvido no Google Colab na seção 4.1, para otimização do problema de *flow shop*. As seções que não estavam presentes na primeira e segunda iterações e foram acrescentadas no decorrer do desenvolvimento estão destacadas por comentários.

```
!pip install -q pyomo
!apt-get install -y -qq coinor-cbc
%matplotlib inline
from __future__ import division
from pyomo.environ import *
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
model = ConcreteModel()

# i é o índice correspondente aos produtos e k, às máquinas.
model.i = Set(initialize = [1,2,3])
model.k = Set(initialize = [1,2,3,4])
model.ij = Set(initialize = model.i * model.i, filter = lambda model,
 i, j: j != i)
model.ik = Set(initialize = model.i * model.k, filter = lambda model,
 i, k: (i,k) != (3,3))

# pzin contém os valores dos parâmetros P.
pzin = {}
pzin[1,1] = 6
pzin[1,2] = 8
pzin[1,3] = 9
pzin[1,4] = 4
pzin[2,1] = 1
pzin[2,2] = 3
pzin[2,3] = 9
pzin[2,4] = 6
pzin[3,1] = 4
pzin[3,2] = 5
pzin[3,3] = 0
pzin[3,4] = 6
```

```
model.P = Param(model.i, model.k, initialize=pzin, default=0)

# Um limite superior para as variáveis (ls) é calculado, sendo a soma
 de duração de todas as operações, se apenas uma máquina pudesse oper
ar de cada vez.
ls = sum([model.P[i, k] for (i,k) in model.ik])

model.T = Var(model.i, model.k, domain=NonNegativeReals,bounds=(0,ls)
)
model.S = Var(model.k, domain=NonNegativeReals,bounds=(0,ls))
model.Y = Var(model.ij, model.k, domain=Boolean)
model.Z = Var(domain=NonNegativeReals,bounds=(0,ls))

# restrição de desigualdade: Tik+1 >= Tik + Pik
# (m-1)n = 9 restrições desse tipo
model.desigualdade = ConstraintList()
for i in model.i:
  for k in [1,2,3]:
    h = k + 1
    model.desigualdade.add(model.T[i,h] >= model.T[i,k] + model.P[i,k
])

# Esta restrição determina que a variável Z seja maior ou igual ao te
mpo máximo do final da operação de todas as máquinas (Z >= Tik + Pik)
.
model.Tmax = ConstraintList()
for i in model.i:
  for k in model.k:
    model.Tmax.add(model.Z >= model.T[i,k] + model.P[i,k])

# Restrição de igualdade: Tik = Sk + sum ( Pjk * Yjik )
# mn = 12 restrições desse tipo.
model.igualdade = ConstraintList()
for k in model.k:
  for i in model.i:
    model.igualdade.add(sum(model.P[j,k] * model.Y[j,i,k] for j in mo
del.i if i != j) + model.S[k] - model.T[i,k] == 0)

# Esta restrição foi adicionada na segunda iteração, pois é impossíve
l que j preceda i e i preceda j numa mesma máquina k.
model.soma = ConstraintList()
for k in model.k:
  for i in model.i:
    for j in model.i:
      if j != i:
```

```
        model.soma.add(model.Y[i,j,k] + model.Y[j,i,k] == 1)

# Estas duas restrições foram acrescentadas na terceira iteração, par
a impedir a ocorrência de loops.
model.loop1 = ConstraintList()
for k in model.k:
  model.loop1.add(model.Y[1,2,k]+model.Y[2,3,k]+model.Y[3,1,k]<=2)
model.loop2 = ConstraintList()
for k in model.k:
  model.loop2.add(model.Y[2,1,k]+model.Y[1,3,k]+model.Y[3,2,k]<=2)

model.obj = Objective(rule=model.Z)

def final(model):
    TransformationFactory('gdp.hull').apply_to(model)
    return model

final(model)

def produtoshop_solve(model):
    SolverFactory('cbc').solve(model)
    model.pprint()
    results = [{'Produto': i,
                'Máquina': k,
                'Início': model.T[i, k](),
                'Duração': model.P[i,k],
                'Término': model.T[(i, k)]() + model.P[i,k]}
               for i,k in model.ik]
    return results

results = produtoshop_solve(model)
results

# Neste bloco, um dataframe é criado para apresentar os resultados ob
tidos na resolução do problema.

schedule = pd.DataFrame(results)

print('\nSequência por Produto')
print(schedule.sort_values(by=['Produto','Início']).set_index(['Produ
to', 'Máquina']))

print('\nSequência por Máquina')
print(schedule.sort_values(by=['Máquina','Início']).set_index(['Máqui
na', 'Produto']))
```

```
# Este bloco configura os gráficos gerados a partir dos resultados.

def visualize(results):

    schedule = pd.DataFrame(results)
    PRODUTOS = sorted(list(schedule['Produto'].unique()))
    MACHINES = sorted(list(schedule['Máquina'].unique()))
    makespan = schedule['Término'].max()

    bar_style = {'alpha':1.0, 'lw':25, 'solid_capstyle':'butt'}
    text_style = {'color':'white', 'weight':'bold', 'ha':'center', 'v
a':'center'}
    colors = mpl.cm.Dark2.colors

    schedule.sort_values(by=['Produto', 'Início'])
    schedule.set_index(['Produto', 'Máquina'], inplace=True)

    fig, ax = plt.subplots(2,1, figsize=(12, 5+(len(PRODUTOS)+len(MAC
HINES))/4))

    for jdx, j in enumerate(PRODUTOS, 1):
        for mdx, m in enumerate(MACHINES, 1):
            if (j,m) in schedule.index:
                xs = schedule.loc[(j,m), 'Início']
                xf = schedule.loc[(j,m), 'Término']
                ax[0].plot([xs, xf], [jdx]*2, c=colors[mdx%7], **bar_
style)
                ax[0].text((xs + xf)/2, jdx, m, **text_style)
                ax[1].plot([xs, xf], [mdx]*2, c=colors[jdx%7], **bar_
style)
                ax[1].text((xs + xf)/2, mdx, j, **text_style)

    ax[0].set_title('Sequência por Produto')
    ax[0].set_ylabel('Produto')
    ax[1].set_title('Sequência por Máquina')
    ax[1].set_ylabel('Máquina')

    for idx, s in enumerate([PRODUTOS, MACHINES]):
        ax[idx].set_ylim(0.5, len(s) + 0.5)
        ax[idx].set_yticks(range(1, 1 + len(s)))
        ax[idx].set_yticklabels(s)
        ax[idx].text(makespan, ax[idx].get_ylim()[0]-
0.2, "{0:0.1f}".format(makespan), ha='center', va='top')
        ax[idx].plot([makespan]*2, ax[idx].get_ylim(), 'r--')
```

```
        ax[idx].set_xlabel('Tempo')
        ax[idx].grid(True)

    fig.tight_layout()

visualize(results)
```

APÊNDICE C – Algoritmo para resolução do problema de alocação de água

Este apêndice contém o código do modelo desenvolvido no Google Colab na seção 4.2, para otimização do problema de alocação de água.

```
!pip install -q pyomo
!wget -N -q "https://ampl.com/dl/open/ipopt/ipopt-linux64.zip"
!unzip -o -q ipopt-linux64

from pyomo.environ import *
import numpy as np
import pandas as pd

model = ConcreteModel()

# i subsistemas
# i = 1 -> fonte de água
# i = 2,3,4 -> subsistemas que utilizam água
# i = 5,6,7 -> subsistemas de tratamento
# i = 8 -> despejo
model.i = RangeSet(1,8)
# k poluentes
model.k = Set(initialize = [1,2,3])

# P[i,k] e r[i,k] são, respectivamente, as taxas de geração e remoção
 do poluente k no subsistema i.
# Poluentes são gerados apenas nos subsistemas i = 2, 3, 4 e removido
s apenas em i = 5, 6, 7.

P_init = {}
for i in [1,5,6,7,8]:
  for k in model.k:
    P_init[i,k] = 0

P_init[2,1] = 0.0179
P_init[2,2] = 0.0005
P_init[2,3] = 0.0012
P_init[3,1] = 0.536
P_init[3,2] = 0.0033
P_init[3,3] = 0.0005
P_init[4,1] = 0.0013
P_init[4,2] = 0.0057
P_init[4,3] = 0.002
```

```
r_init = {}
for i in [1,2,3,4,8]:
  for k in model.k:
    r_init[i,k] = 0

r_init[5,1] = 0.999
r_init[5,2] = 0
r_init[5,3] = 0
r_init[6,1] = 0
r_init[6,2] = 0.95
r_init[6,3] = 0.20
r_init[7,1] = 0.9
r_init[7,2] = 0.9
r_init[7,3] = 0.97

# z[i,k] é a máxima quantidade de poluente k permitida na alimentação
 do subsistema i em ppm.

z_init = {}
z_init[2,1] = 0
z_init[2,2] = 0
z_init[2,3] = 0
z_init[3,1] = 500
z_init[3,2] = 20
z_init[3,3] = 50
z_init[4,1] = 20
z_init[4,2] = 120
z_init[4,3] = 50
z_init[8,1] = 2
z_init[8,2] = 2
z_init[8,3] = 5

model.P = Param(model.i, model.k, initialize=P_init)
model.r = Param(model.i, model.k, initialize=r_init)
model.z = Param(model.i, model.k, initialize=z_init, default=0)

model.x = Var(model.i, model.k, domain=NonNegativeReals, bounds=[0,1]
)
model.d = Var(model.i, model.i, domain=NonNegativeReals, bounds=[0,1]
)
model.Q = Var(model.i, domain=PositiveReals, bounds=[0.00001, 1000])

# Balanço de massa para a água.
model.BMW = ConstraintList()
```

```
for i in model.i:
  if i == 1:
    continue
  else:
    model.BMW.add(sum(model.d[i,j] * model.Q[j] for j in model.i if i
 > j) == model.Q[i])

# Balanço de massa para o poluente k.
model.BMX = ConstraintList()
for k in model.k:
  for i in model.i:
    if i == 1:
      continue
    else:
      model.BMX.add( (1 - model.r[i,k]) * sum(model.d[i,j] * model.Q[
j] * model.x[j,k] for j in model.i if i > j) + model.P[i,k] == model.
Q[i] * model.x[i,k] )

# Restrição da máxima quantidade de poluente em cada subsistema.
model.BMZ = ConstraintList()
for k in model.k:
  for i in [2,3,4,8]:
    if i == 1:
      continue
    else:
      model.BMZ.add( sum(model.d[i,j] * model.Q[j] * model.x[j,k] for
 j in model.i if i > j)/model.Q[i] <= model.z[i,k] / 1000000 )

model.Q2 = Constraint(rule=model.Q[2] == 45.8)
model.Q3 = Constraint(rule=model.Q[3] == 32.7)
model.Q4 = Constraint(rule=model.Q[4] == 56.5)

# Balanço de massa global do sistema.
model.BM18 = Constraint(rule=model.Q[1] == model.Q[8])

# Soma das razões de separação deve ser 1.
model.delta = ConstraintList()
for j in model.i:
  if j == 8:
    model.delta.add( sum(model.d[i,j] for i in model.i if i != j) ==
0 )
  else:
    model.delta.add( sum(model.d[i,j] for i in model.i if i != j) ==
1 )
```

```
def op_cost(model):
  return 8000*( 1*model.Q[5] + 0.0067*model.Q[7] )

def water_cost(model):
  return model.Q[1] * 0.3 * 8000

def invest_cost(model):
  return ( 16800*model.Q[5]**0.7 + 4800*model.Q[6]**0.7 + 12600*model
.Q[7]**0.7 ) / 7.5

def obj_expression(model):
  return op_cost(model) + water_cost(model) + invest_cost(model)
model.OBJ = Objective(rule=obj_expression)

SolverFactory('ipopt', executable='/content/ipopt').solve(model).writ
e()

model.pprint()
```

APÊNDICE D – Quantidade de poluente no efluente do subsistema *i*

Subsistema	Poluente	x_{ij} [ppm]
1	1	0
1	2	0
1	3	0
2	1	391
2	2	11
2	3	26
3	1	16782
3	2	112
3	3	41
4	1	26
4	2	101
4	3	36
5	1	12
5	2	84
5	3	37
6	1	22
6	2	5
6	3	29
7	1	2
7	2	2
7	3	1
8	1	2
8	2	2
8	3	1