ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC ỨNG DỤNG

MÔ HÌNH HÓA TOÁN HỌC - CO2011

Bài tập lớn

"Cutting Stock Problem"

Instructor(s): Mai Xuân Toàn

Students: Trịnh Quốc Bảo - 2210284 *(Group DT01 - Team 06, **Leader**)*
Ngô Trường Bách - 2210183 *(Group DT01 - Team 06)*
Lê Đức Cường - 2210423 *(Group DT01 - Team 06)*
Nguyễn Hạo Duy - 2210512 *(Group DT01 - Team 06)*
Hoàng Thanh Chí Bảo - 2210205 *(Group DT01 - Team 06)*

HO CHI MINH CITY, JULY 2024

Mục lục

Danh sách các ký hiệu

$\mathbb{N}$ Tập hợp các số tự nhiên

$\mathbb{R}$ Tập hợp các số thực

$\mathbb{R}^+$ Tập hợp các số thực dương

Danh sách các từ viết tắt

CSP The Cutting-Stock Problem

SHP Sequential Heuristic Procedure

RMP Restricted Master Problem

RLPM Restricted Linear Programming Master Problem

LMP Linear Master Programming

MILP Mixed Integer Linear Programming

Fig. Figure

Tab. Table

Sys. System of Equations

Eq. Equation

e.g. For Example

i.e. That Is

Danh sách hình ảnh

Danh sách bảng

Danh sách thành viên & khối lượng công việc

STT	Họ và tên	MSSV	Công việc	Ghi chú
1	Trịnh Quốc Bảo	2210284	- Giải thuật Column generation - Viết LaTeX	100%
2	Ngô Trường Bách	2210183	- Giải thuật Column generation - Viết LaTeX	100%
3	Lê Đức Cường	2210423	- Giải thuật Optimal cutting known patterns - Viết LaTeX	100%
4	Nguyễn Hạo Duy	2210512	- Giải thuật Optimal cutting known patterns - Viết LaTeX	100%
5	Hoàng Thanh Chí Bảo	2210205	- Giải thuật Optimal cutting known patterns - Viết LaTeX	100%

Bảng 1: Danh sách thành viên & khối lượng công việc

1 Lời cảm ơn

Mô hình hóa toán học là một môn học cơ bản về các kỹ thuật mô hình hóa thông qua các kiến thức về logic vị từ, quy hoạch tuyến tính, automata và ngôn ngữ hình thức. Môn học cũng cung cấp cho sinh viên kỹ năng cơ bản để giải quyết các bài toán thực tiễn bằng cách chuyển các vấn đề thành mô hình và giải các mô hình đó. Với đề tài được phân công **"Cutting-stock problem"**, nhóm chúng em đã phân chia công việc hợp lý và trách nhiệm của mỗi thành viên đối với bài tập lớn lần này.

Cùng với việc thực hiện đề tài này, thay mặt nhóm em xin gửi lời cảm ơn sâu sắc đến thầy Mai Xuân Toàn đã nhiệt tình chỉ dẫn các bước, hướng nghiên cứu, thực hiện cũng như yêu cầu cần có của đề tài trong suốt thời gian thực hiện đề tài. Thầy đã không chỉ cung cấp cho chúng em những kiến thức quý báu mà còn truyền đạt những kinh nghiệm thực tiễn, giúp chúng em hiểu rõ hơn về cách áp dụng lý thuyết vào thực tế. Sự tận tâm và nhiệt huyết của thầy đã là nguồn động lực lớn lao để chúng em hoàn thành tốt đề tài này.

Trong quá trình thực hiện, dù đã rất cố gắng nhưng không tránh khỏi những sai sót và hạn chế nhất định. Chúng em đã học được rất nhiều từ những lỗi lầm đó và hy vọng rằng với sự góp ý, bổ sung thêm của thầy, đề tài của chúng em sẽ được hoàn thiện hơn.

Chúng em xin chân thành cảm ơn và mong rằng sẽ tiếp tục nhận được sự hướng dẫn và hỗ trợ từ thầy trong những dự án và nghiên cứu tiếp theo.

2 Giới thiệu vấn đề

Bài toán Cutting-Stock có rất nhiều ứng dụng trong việc lập kế hoạch sản xuất ở nhiều ngành công nghiệp như luyện kim, nhựa, giấy,.... Nhìn chung, các bài toán dạng Cutting-Stock thường dựa vào việc cắt các mảnh nguyên liệu lớn không giới hạn có chiều dài c thành một tập các mảnh nhỏ hơn theo yêu cầu số lượng v_i và với chiều dài w_i đồng thời tối ưu một hàm mục tiêu nào đó. Hàm mục tiêu đó có thể là hàm giúp giảm thiểu sự hao phí, tối ưu hóa lợi nhuận, giảm thiểu giá thành, giảm thiểu số sản phẩm được sử dụng,...

Các bài toán dạng CSP(*"Cutting-stock problem"*) có thể dễ dàng mô hình hóa và lập ra được công thức. Tuy nhiên, những bài toán dạng này lại thuộc lớp NP-hard nên nó rất khó giải quyết một cách tối ưu. Vậy nên, việc tìm được cách giải tối ưu nhất có thể sẽ giúp giảm thiểu chi phí sản xuất một cách đáng kể trong các nhà máy.

Ở trong tài liệu này, nhóm chúng em sẽ tiến hành tim hiểu hai thuật toán để giải quyết bài toán CSP đồng thời so sánh độ hiệu quả của chúng.

Bố cục của phần còn lại trong tài liệu này sẽ như sau: Ở trong mục 3, chúng em sẽ trình bày lại về lịch sử của những nghiên cứu đã được công bố về bài toán CSP. Mục 4, chúng em sẽ trình bày định nghĩa toán học của bài toán 1D Cutting-Stock cổ điển. Ở mục 5, chúng em sẽ trình bày hai giải thuật đồng thời đưa ra ví dụ thực tế mà ở đó việc giải quyết bài toán CSP dẫn đến giải quyết vấn đề ngoài đời thực. Ở mục 6, chúng em sẽ phân tích kết quả, giải thích lời giải tối ưu và so sánh độ hiệu quả của hai giải thuật. Mục cuối cùng sẽ là phần kết luận và thảo luận hướng đi tiếp theo trong việc giải bài toán 1D CSP.

3 Literature review

Bài toán CSP lần đầu tiên được trình bày vào năm 1939 bởi Kantorovich . Tiến bộ quan trọng nhất trong việc giải quyết bài toán CSP một chiều là công trình sáng tạo của Gilmore và Gomory , nơi họ đề xuất phương pháp tạo mẫu trỳ hoãn để giải quyết vấn đề bằng cách sử dụng lập trình tuyến tính. Dyckhoff phân loại các giải pháp của bài toán CSP thành hai nhóm: định hướng mẫu và định hướng vật phẩm. Hơn nữa, ông phân loại các vấn đề cắt bằng cách sử dụng bốn đặc điểm: tính chiều, loại phân công, sự phối hợp của các đối tượng lớn, sự phối hợp của các đối tượng nhỏ. Waescher và Gau kết luận rằng các giải pháp số nguyên tối ưu có thể đạt được trong hầu hết các trường hợp. Gradisar et al. trình bày Sequential Heuristic Procedure (SHP) để tối ưu hóa bài toán CSP 1 chiều khi tất cả các chiều dài là khác nhau. Hơn nữa, họ đề xuất một phương pháp bằng cách kết hợp phương pháp LP định hướng mẫu và quy trình **SHP** định hướng vật phẩm Vance et al và Valerio de Carvalho đã trình bày một số cố gắng kết hợp tạo cột và nhánh-và-cận (branch and bound). Họ đã có thể đạt được các giải pháp chính xác cho các trường hợp CSP khá lớn. Scheithauer et al. đã trình bày một thuật toán cắt mặt phẳng để giải quyết chính xác CSP 1 chiều. Mukhacheva et al. đã đề xuất một phương pháp nhánh-và-cận sửa đổi cho CSP 1 chiều. Belov và Scheithauer đã đề xuất một phương pháp kết hợp một thuật toán cắt mặt phẳng với tạo cột cho CSP 1 chiều với nhiều chiều dài . Umetani et al. xem xét rằng số lượng mẫu cắt khác nhau trong CSP 1 chiều bị giới hạn trong một miền giới hạn cho trước. Sau đó, họ đề xuất một phương pháp dựa trên "metaheuristics" và kết hợp một phương pháp tạo mẫu thích ứng. Johnston và Sadinlija đã tạo ra một mô hình mới giải quyết tính phi tuyến trong bài toán CSP 1 chiều, giữa các biến mẫu và độ dài chạy mẫu bằng cách sử dụng biến 0-1. Belov và Scheithauer đã phát triển một thuật toán nhánh-và-cắt-và-giá (branch-and-cut-and-price) cho việc cắt kho một chiều và cắt hai giai đoạn hai chiều. Họ đã nghiên cứu sự kết hợp của sự nới lỏng bài toán tuyến tình (LP relaxation) tại mỗi nút nhánh-và-giá được củng cố bởi các cắt Chvatal-Gomory và Gomory hỗn hợp-số nguyên. Trong những năm gần đây, đã có một số nỗ

lực giải quyết vấn đề này bằng cách sử dụng các cách tiếp cận khác nhau. Dikili et al. đã trình bày một phương pháp loại bỏ tuần tự để giải quyết 1D-CSP trong sản xuất tàu, bằng cách sử dụng các mẫu cắt thu được bằng các phương pháp phân tích ở giai đoạn mô hình hóa toán học. Reinertsen và Vossen đã xem xét CSP khi các đơn hàng có ngày đến hạn và đề xuất các mô hình tối ưu hóa mới và các quy trình giải quyết để giải quyết vấn đề này. Cherri et al. đã nghiên cứu vấn đề cắt kho với các phần thừa sử dụng được và sửa đổi các "heuristic" nổi tiếng trong tài liệu để giải quyết vấn đề này. Hơn nữa, Cherri et al. giả định rằng các phần còn lại có sẵn trong kho có ưu tiên sử dụng trong quá trình cắt và phát triển các heuristic của họ xem xét các ưu tiên này. Berberler và Nuriyev đã xem xét vấn đề tập hợp con như một vấn đề phụ để giải quyết 1D-CSP và đề xuất một "heuristic" dựa trên lập trình động. Mobasher và Ekici đã nghiên cứu một trường hợp tổng quát hơn của CSP cổ điển, được gọi là vấn đề cắt kho với chi phí thiết lập, trong đó mục tiêu là tối thiểu hóa tổng chi phí sản xuất bao gồm cả chi phí vật liệu và chi phí thiết lập. Họ đã phát triển một mô hình tuyến tính số nguyên hỗn hợp và đề xuất một thuật toán cho một trường hợp đặc biệt của vấn đề.

4 Bài toán One-dimensional cutting-stock

Bài toán One-dimensional cutting-stock (ngắn gọn là 1D CSP) là một lớp bài toán thuộc NP-hard [6] thường xảy ra trong quá trình sản xuất của nhiều ngành công nghiệp như: sắt, quần áo, aluminum,... và đã thu hút sự chú ý trên toàn thế giới.

Bài toán 1D CSP là một dạng của bài toán quy hoạch tuyến tính với một biến quyết định cho mỗi trường hợp có thể xảy ra. Nếu số lượng yêu cầu nhỏ thì số lượng trường hợp sẽ đủ nhỏ để có thể giải quyết bằng một giải thuật đặc trưng. Tuy nhiên cũng có trường hợp số lượng trường hợp diễn tiến theo cấp số nhân nếu số lượng yêu cầu lớn. Trong những trường hợp như vậy, cần phải có những cách tiếp cận khác.

Trong bài toán 1D CSP, ta cắt mảnh nguyên liệu có chiều dài cố định thành những mảnh nhỏ hơn theo yêu cầu đồng thời giảm thiểu hao phí. Trong đó:

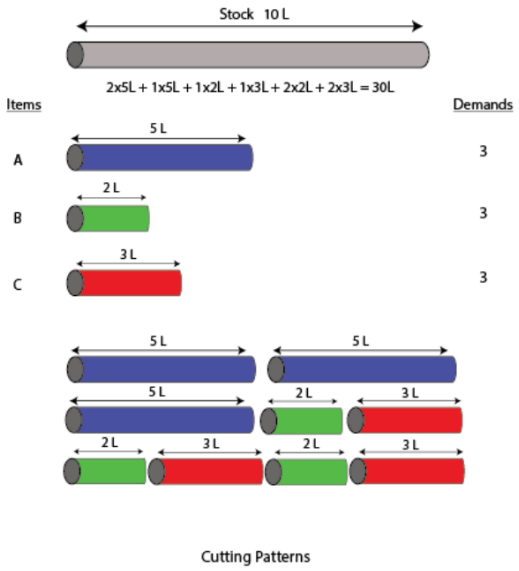
- c là chiều dài của (các) mảnh nguyên liệu được cắt.
- w_i là chiều dài của các mảnh được yêu cầu với i = 1,2,...,n.
- v_i là số lượng yêu cầu của mảnh nguyên liệu i
- x_{ij} là số mảnh nguyên liệu i được cắt từ mảnh nguyên liệu lớn j
- y_j kí hiệu mảnh nguyên liệu lớn j có được dùng trong kế hoạch cắt hay không($y_j = 1$ nếu được cắt) với j = 1,2,...,m
- t_j kí hiệu chiều dài còn lại của mảnh nguyên liệu lớn j

Hàm mục tiêu là hàm dùng để tối thiểu độ hao phí. Từ những kí hiệu trên ta có công thức

như sau:

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{m} t_j \\
\text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} w_j + t_j = c_j, && j = 1, \ldots, m \\
& \sum_{i=1}^{n} x_{ij} = v_i, && i = 1, \ldots, n \\
& x_{ij} \geq 0 \text{ integer}, && i = 1, \ldots, n; j = 1, \ldots, m \\
& t_j \geq 0 \text{ integer}, && j = 1, \ldots, m \\
& y_j \in \{0, 1\}, && j = 1, \ldots, m
\end{aligned}
$$

Ở trong mô hình này, ràng buộc đầu tiên dùng để tính toán sự hao phí của của nguyên liệu trong quá trình cắt, ràng buộc thứ hai đảm bảo rằng những yêu cầu đều đã được thỏa mãn đồng thời hàm mục tiêu đạt giá trị nhỏ nhất có thể



Hình 1: 1D CSP với một mảnh nguyên liệu

5 Giải thuật

5.1 Column generation

5.1.1 Dẫn nhập

Ta kí hiệu tập các kích thước có thể cắt được từ thanh nguyên liệu là $i \in 1, \ldots, I$. Mỗi mảnh i có chiều dài w_i, và số lượng yêu cầu là d_i. Chiều dài của thanh nguyên liệu là W.

Mục tiêu của ta là tìm ra số thanh nguyên liệu ít nhất cần phải cắt sao cho thỏa các yêu cầu đặt hàng.

Ta xét một bài toán cụ thể với chiều dài thanh kim loại là $W = 100$ và các yêu cầu còn lại như hình dưới:

```
Data for the cutting stock problem:
  W = 100.0
with pieces:
   i   w_i d_i
  ------------
   1  75.0  38
   2  75.0  44
   3  75.0  30
   4  75.0  41
   5  75.0  36
   6  53.8  33
   7  53.0  36
   8  51.0  41
   9  50.2  35
  10  32.2  37
  11  30.8  44
  12  29.8  49
  13  20.1  37
  14  16.2  36
  15  14.5  42
  16  11.0  33
  17   8.6  47
  18   8.2  35
  19   6.6  49
  20   5.1  42
```

Hình 2: 20 đơn đặt với w_i và d_i tương ứng

Để mô hình bài toán này dưới dạng MILP (Mixed Integer Linear Programming), ta giả định có một tập các thanh nguyên liệu $j = 1, \ldots, J$ để cắt. Đồng thời, ta có hai decision variables (biến quyết định):

- $x_{ij} \geq 0,$ Integer $\forall i = 1, \ldots, I, \forall j = 1, \ldots, J$
- $y_j \in \{0, 1\} \forall j = 1, \ldots, J$

Trong đó:

- x_{ij} là số lượng mảnh có kích thước i được cắt từ thanh nguyên liệu j.
- y_j là biến cho biết ta có dùng thanh j hay không

Ta có công thức cho bài toán trên như sau:

$$\min \sum_{j=1}^{J} y_j \tag{1}$$

$$\text{s.t.} \sum_{i=1}^{N} x_{ij} = W_j y_j, \quad \forall j = 1, \ldots, J \tag{2}$$

$$\sum_{j=1}^{J} x_{ij} = d_i, \quad \forall i = 1, \ldots, I \tag{3}$$

$$x_{ij} \geq 0, \quad \forall i = 1, \ldots, N; j = 1, \ldots, J \tag{4}$$

$$x_{ij} \in \mathbb{Z}, \quad \forall i = 1, \ldots, I; j = 1, \ldots, J \tag{5}$$

$$y_j \in \{0, 1\}, \quad \forall j = 1, \ldots, J \tag{6}$$

Hàm mục tiêu dùng để minimize số thanh nguyên liệu mà ta dùng. Điều kiện (2) để đảm bảo rằng số mảnh cắt được từ thanh nguyên liệu có chiều dài khi cộng lại không thể lớn hơn một thanh nguyên liệu. Điều kiện (3) đảm bảo ta thỏa mãn được yêu cầu của đơn hàng.

Ta có thể dùng juMP - một mô hình ngôn ngữ cho việc tối ưu toán học trong Julia để thử giải quyết:

```
I = length(data.pieces)
J = 1_000  # Some large number
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:I, 1:J] >= 0, Int)
@variable(model, y[1:J], Bin)
@objective(model, Min, sum(y))
@constraint(model, [i in 1:I], sum(x[i, :]) >= data.pieces[i].d)
@constraint(
    model,
    [j in 1:J],
    sum(data.pieces[i].w * x[i, j] for i in 1:I) <= data.W * y[j],
);
```

Hình 3: Đoạn code mô tả lại công thức trên

```
set_time_limit_sec(model, 5.0)
optimize!(model)
solution_summary(model)
```

Hình 4: Tiến hành giải công thức trên với giới hạn thời gian

```
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : TIME_LIMIT
  Message from the solver:
  "kHighsModelStatusTimeLimit"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : NO_SOLUTION
  Objective value    : 4.11000e+02
  Objective bound    : 2.93000e+02
  Relative gap       : 2.87105e-01

* Work counters
  Solve time (sec)   : 5.04147e+00
  Simplex iterations : 20471
  Barrier iterations : -1
  Node count         : 0
```

Hình 5: Kết quả

Ta không nhận được kết quả trong thời gian cho phép. Như vậy, việc dùng cách tiếp cận này là không khả thi khi gặp những trường hợp thực tế bởi vì phương pháp mất quá nhiều thời gian để giải. Điều này dẫn đến nhu cầu phải đề ra một công thức, phương pháp khác nhanh hơn là **Column generation**.

5.1.2 Lý thuyết Column generation

5.1.2.a Giới thiệu về Column generation

Column Generation là một kỹ thuật có thể được sử dụng để giải các bài toán tối ưu hóa tuyến tính lớn bằng cách chỉ tạo ra những biến có ảnh hưởng đến hàm mục tiêu. Đây là yếu tố quan trọng đối với các bài toán lớn với nhiều biến vì các kỹ thuật này giúp đơn giản hóa việc xây dựng bài toán, khi không cần phải liệt kê tất cả khả năng xảy ra[4].

5.1.2.b Lý thuyết, Phương pháp và Giải thuật

Lý thuyết

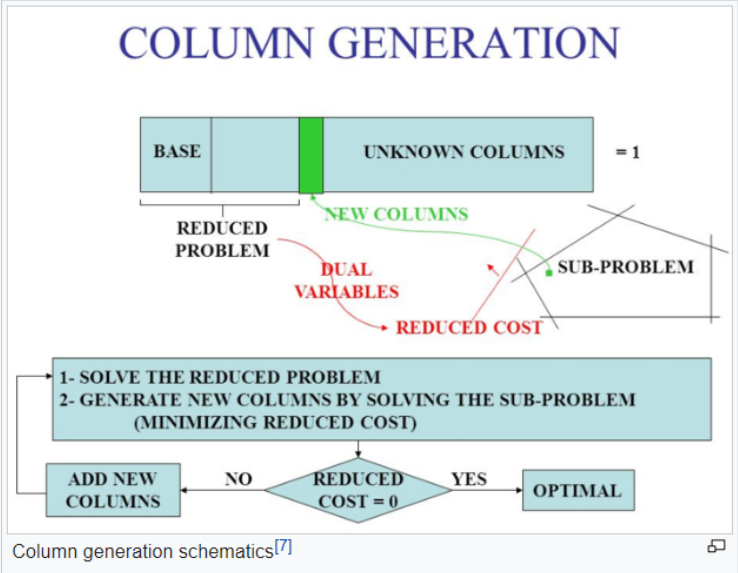
Phương pháp này hoạt động như sau: đầu tiên, bài toán gốc cần được chia thành hai phần: bài toán chính và bài toán con.

- Bài toán chính là việc biểu diễn ban đầu của bài toán theo hướng từng cột (tức là xét từng cột một) với chỉ một tập con các biến được xem xét.
- Bài toán con là một bài toán mới được tạo ra nhằm xác định một biến mới có triển vọng. Hàm mục tiêu của bài toán con là chi phí giảm của biến mới so với các biến đối ngẫu hiện tại, và các ràng buộc yêu cầu biến đó phải tuân theo các ràng buộc tự nhiên có sẵn. Bài toán con cũng được gọi là RMP (Restricted Master Problem), từ điều này, có thể suy ra rằng phương pháp này rất phù hợp với các bài toán có tập ràng buộc cho phép phân rã tự nhiên (tức là phân tích) thành các hệ thống con, đại diện cho các cấu trúc tổ hợp đã được hiểu rõ.

Để thực hiện phân rã từ bài toán gốc thành bài toán chính và bài toán con, có nhiều kỹ thuật khác nhau. Lý thuyết đứng sau phương pháp này dựa trên phân rã Dantzig-Wolfe.

Tóm lại, khi bài toán chính được giải, ta có thể thu được các giá đối ngẫu cho mỗi ràng buộc trong bài toán chính. Thông tin này sau đó được sử dụng trong hàm mục tiêu của bài toán con. Bài toán con được giải. Nếu giá trị hàm mục tiêu của bài toán con là âm, thì một biến với chi phí giảm âm phải được xác định. Biến này sau đó được thêm vào bài toán chính, và bài toán chính được giải lại. Việc giải lại bài toán chính sẽ tạo ra một tập giá đối ngẫu mới, và quá trình này lặp lại cho đến khi không còn biến nào có chi phí giảm âm được xác định. Nếu bài toán con trả về một nghiệm với chi phí giảm không âm, ta có thể kết luận rằng nghiệm của bài toán chính là tối ưu.

Phương Pháp



Hình 6: Sơ đồ Column generation

Xem xét bài toán dưới dạng sau:

$$(IP) z = \max \left\{ \sum_{k=1}^{K} c^k x^k : \sum_{k=1}^{K} A^k x^k = b,\ x^k \in X^k \text{ với } k = 1, \ldots, K \right\}$$

$$\text{Với } X^k = \left\{ x^k \in \mathbb{Z}_+^{n_k} : D^k x^k \leq d^k \right\} \text{ cho } k = 1, \ldots, K$$

Giả sử mỗi tập hợp X^k chứa một tập hợp điểm lớn nhưng hữu hạn $\left\{x^{k,t}\right\}_{t=1}^{T_k}$ thì ta có:

$$X^k = \left\{ x^k \in \mathbb{R}^{n_k} \mid x^k = \sum_{t=1}^{T_k} \lambda_{k,t} x^{k,t},\ \sum_{t=1}^{T_k} \lambda_{k,t} = 1,\ \lambda_{k,t} \in \{0,1\} \text{ cho } k = 1, \ldots, K \right\}$$

Lưu ý rằng, dưới giả định mỗi tập hợp X^k đều bị chặn bởi k = 1, . . . , K, phương pháp sẽ giải một bài toán tương đương như sau:

$$\max \left\{ \sum_{k=1}^{K} \gamma^k \lambda^k : \sum_{k=1}^{K} B^k \lambda^k = \beta,\ \lambda^k \geq 0 \text{ cho } k = 1, \ldots, K \right\}$$

Trong đó, mỗi ma trận B^k có số cột rất lớn, mỗi cột tương ứng với một điểm khả thi trong X^k và mỗi vector λ^k chứa các biến tương ứng.

Bây giờ, thay thế x^k dẫn đến một bài toán Master Problem (LPM) tương đương:

$$\begin{array}{rl} z = & \max \sum_{k=1}^{K} \sum_{t=1}^{T_k} \left(c^k x^{k,t}\right) \lambda_{k,t} \\ \text{(IPM)} & \sum_{k=1}^{K} \sum_{t=1}^{T_k} \left(A^k x^{k,t}\right) \lambda_{k,t} = b \\ & \sum_{t=1}^{T_k} \lambda_{k,t} = 1 \text{ cho } k = 1, \ldots, K \\ & \lambda_{k,t} \in \{0,1\} \text{ cho } t = 1, \ldots, T_k \text{ và } k = 1, \ldots, K. \end{array}$$

Để giải quyết bài toán chính (Master Linear Programming), nhóm chúng tôi sẽ sử dụng thuật toán sinh cột (Column Generation). Điều này giúp giải quyết bài toán nới lỏng của một chương trình tuyến tính được gọi là Bài toán Lập trình Tuyến tính Master (LPM):

$$z^{LPM} = \max \sum_{k=1}^{K} \sum_{t=1}^{T_k} \left(c^k x^{k,t}\right) \lambda_{k,t}$$

$$\text{s.t.} \sum_{k=1}^{K} \sum_{t=1}^{T_k} \left(A^k x^{k,t}\right) \lambda_{k,t} = b$$

$$\sum_{t=1}^{T_k} \lambda_{k,t} = 1 \text{ cho } k = 1, \ldots, K$$

$$\lambda_{k,t} \geq 0 \text{ cho } t = 1, \ldots, T_k \text{ và } k = 1, \ldots, K$$

Tại mỗi cột $\begin{pmatrix} c^k x \\ A^k x \\ e_k \end{pmatrix}$ tương ứng với mỗi $x \in X^k$. Trong các bước tiếp theo của phương pháp này, chúng tôi sẽ sử dụng $\{\pi_i\}_{i=1}^{m}$ làm các biến liên hợp liên quan đến các ràng buộc chung, và $\{\mu_k\}_{k=1}^{K}$ làm các biến liên hợp cho tập hợp ràng buộc thứ hai. Các biến này cũng được gọi là các ràng buộc về tính lồi. Ý tưởng là giải bài toán lập trình tuyến tính bằng phương pháp đơn hình nguyên thủy. Tuy nhiên, bước định giá để chọn cột gia nhập cơ sở cần được điều chỉnh vì số lượng cột rất lớn. Thay vì định giá từng cột một, việc tìm cột có giá giảm lớn nhất thực tế là một tập hợp các bài toán tối ưu hóa K.

Khởi tạo: chúng tôi giả định rằng một tập hợp con của các cột (ít nhất một cột cho mỗi k) có sẵn, cung cấp một bài toán Lập trình Tuyến tính Master Bị Hạn (RLPM) khả thi:

$$z^{LPM} = \max \tilde{c}\tilde{\lambda}$$

$$\tilde{A}\tilde{\lambda} = b$$

$$\tilde{\lambda} \geq 0$$

trong đó $\tilde{b} = \begin{pmatrix} b \\ 1 \end{pmatrix}$, $\tilde{A}$ được tạo ra từ tập hợp các cột có sẵn và $\tilde{c}$ và $\tilde{\lambda}$là các chi phí và biến tương ứng. Giải bài toán RLPM cung cấp một nghiệm nguyên thủy tối ưu $\tilde{\lambda}^*$ và một nghiệm đối tối ưu $(\pi, \mu) \in \mathbb{R}^m \times \mathbb{R}^k$.

Tính khả thi nguyên thủy: Bất kỳ nghiệm khả thi nào của RLMP đều là nghiệm khả thi của LPM. Cụ thể hơn, $\tilde{\lambda}^*$ là một nghiệm khả thi của LPM, và do đó,

$$\tilde{z}^{LPM} = \tilde{c}\tilde{\lambda}^* = \sum_{i=1}^{m} \pi_i b_i + \sum_{k=1}^{K} \mu_k \leq z^{LPM}$$

Kiểm tra tính tối ưu cho LPM: Cần kiểm tra xem (π, μ) có phải là nghiệm đối khả thi cho LPM không. Điều này có nghĩa là cần kiểm tra cho mỗi cột, tức là cho mỗi k và cho mỗi $x \in X^k$ xem giá giảm có thỏa mãn $c^k x - \pi A^k x - \mu_k \leq 0$ không. Thay vì kiểm tra từng điểm một, chúng ta xử lý tất cả các điểm trong X^k một cách ngầm định bằng cách giải một bài toán tối ưu hóa con:

$$\zeta_k = \max\left\{\left(c^k - \pi A^k\right)x - \mu_k \ : \ x \in X^k\right\}.$$

Tiêu chí dừng: Nếu $\zeta_k > 0$ cho k = 1, ..., K thì nghiệm (π, μ) là nghiệm đối khả thi cho LPM, và do đó

$$z^{LPM} \leq \sum_{i=1}^{m} \pi_i b_i + \sum_{k=1}^{K} \mu_k.$$

Vì giá trị của nghiệm nguyên thủy khả thi $\tilde{\lambda}$ bằng với giá trị của ràng buộc trên, $\tilde{\lambda}$ là tối ưu cho LPM.

Tạo cột mới: Nếu $\zeta_k > 0$ cho một số k, cột tương ứng với nghiệm tối ưu $\tilde{x}^k$ của bài toán con có giá giảm dương. Việc giới thiệu cột $\begin{pmatrix} c^k x \\ A^k x \\ e_k \end{pmatrix}$ dẫn đến một bài toán Lập trình Tuyến tính Master Bị Hạn có thể dễ dàng được tối ưu hóa lại (ví dụ: bằng phương pháp đơn hình nguyên thủy).

5.1.3 Áp dụng vào bài toán ban đầu

Ta nhận thấy rằng các cột khả thi của biến trong ma trận x mã hóa những cutting patterns. Ví dụ: ta xét thanh nguyên liệu j = 1, thì nghiệm khả thi là:

- $x_{1,1} = 1$ (1 đơn vị của mảnh số 1) với $w_1 = 75.0$
- $x_{13,1} = 1$ (1 đơn vị của mảnh số 13) với $w_{13} = 20.1$
- Các cột còn lại thì $x_{i,1} = 0$

Một ví dụ khác:

- $x_{20,1} = 19$ (19 đơn vị của mảnh số 20) với $w_{19} = 6.6$
- Các cột còn lại thì $x_{i,1} = 0$

Những cutting patterns kiểu như $x_{1,1} = 1$ và $x_{2,1} = 1$ là không khả thi vì tổng chiều dài vượt quá chiều dài của thanh nguyên liệu W.

Như vậy, sẽ có một số hữu hạn cách để ta có thể cắt thanh nguyên liệu thành những cutting patterns thỏa ràng buộc. Ta gọi tập các cutting patterns khả thi là $p = 1, \ldots, P$,với $a_{i,p}$ nghĩa là số mảnh i cắt ở pattern p. Do đó, ta có công thức như sau:

$$\min \sum_{p=1}^{P} x_p \tag{7}$$

$$\text{s.t.} \sum_{p=1}^{P} a_{ip} x_p \geq d_i, \quad \forall i = 1, \ldots, I \tag{8}$$

$$x_p \geq 0, \quad \forall i = 1, \ldots, P \tag{9}$$

$$x_p \in \mathbb{Z}, \quad \forall p = 1, \ldots, P \tag{10}$$

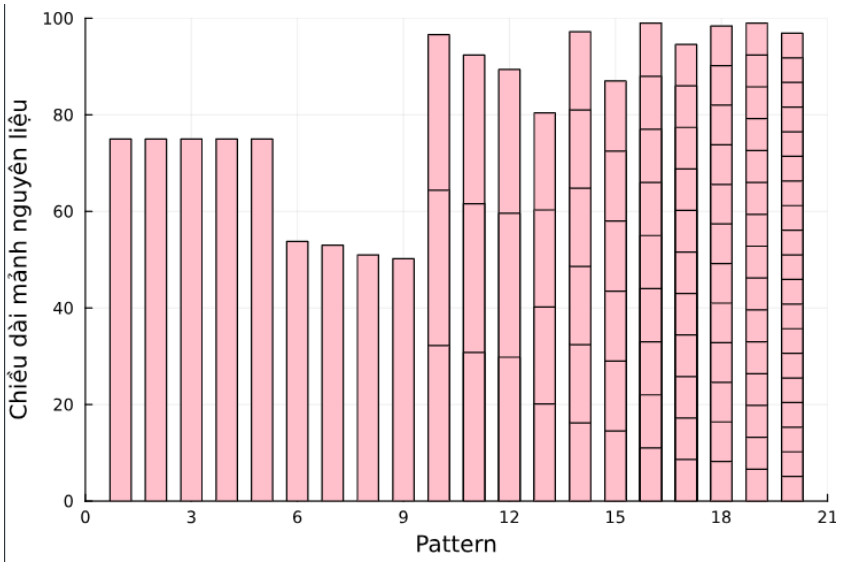
Như đã trình bày ở phần trên, ta sẽ bắt đầu giải bài toán này với một tập nhỏ các patterns ban đầu. Sau đó sẽ dần dần chọn những cột mới để thêm vào bài toán chính (master problem) rồi từ đó sẽ tìm được nghiệm tối ưu.

Với tập các patterns ban đầu, ta chọn cách cắt đơn giản nghĩa là cắt mảnh i cho tới khi thỏa yêu cầu.

```
20-element Vector{SparseArrays.SparseVector{Int64, Int64}}:
  [1]  =  1
  [2]  =  1
  [3]  =  1
  [4]  =  1
  [5]  =  1
  [6]  =  1
  [7]  =  1
  [8]  =  1
  [9]  =  1
  [10]  =  3
  [11]  =  3
  [12]  =  3
  [13]  =  4
  [14]  =  6
  [15]  =  6
  [16]  =  9
  [17]  =  11
  [18]  =  12
  [19]  =  15
  [20]  =  19
```

Hình 7: Tập các patterns ban đầu

Ví dụ với pattern đầu tiên, ta chỉ có thể cắt 1 mảnh số 1 để thỏa yêu cầu; chỉ có thể cắt 3 mảnh số 10 để thỏa yêu cầu. Hình minh họa trực quan của bảng trên:



Hình 8: Minh họa trực quan của tập các patterns ban đầu

Sau khi đã chọn được tập các patterns ban đầu, ta tiến hành tối ưu bài toán ban đầu:

```
# setup model
model = Model(HiGHS.Optimizer)
# to avoid unnecessary output
set_silent(model)
# define variables
@variable(model, x[1:length(patterns)] >= 0, Int)
@objective(model, Min, sum(x))
@constraint(model, demand[i in 1:I], patterns[i]' * x >= data.pieces[i].d)
optimize!(model)
@assert is_solved_and_feasible(model)
# in kết quả ra console
solution_summary(model)
```

Hình 9: Đoạn code dùng để tối ưu

```
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : NO_SOLUTION
  Objective value    : 4.21000e+02
  Objective bound    : 4.21000e+02
  Relative gap       : 0.00000e+00

* Work counters
  Solve time (sec)   : 4.10080e-04
  Simplex iterations : 0
  Barrier iterations : -1
  Node count         : 0
```

Hình 10: Kết quả

Nghiệm của bài toán trên là để thỏa yêu cầu cần **421** thanh nguyên liệu. Nghiệm này chưa được tối ưu bởi vì bài toán mà ta vừa tiến hành giải chỉ ở trong một tập nhỏ của tập đầy đủ các patterns khả thi.

Giờ ta tiến hành chọn cột để thêm vào nhằm cải thiện nghiệm của bài toán. Ta chọn cột mới bằng cách **nới lỏng** điều kiện số nguyên ở ma trận x và xét dual variable (biến đối ngẫu) π_i gắn với yêu cầu ràng buộc i. Ta có các giá trị đối ngẫu của các dual variables:

```
Dual value for demand[1]: 1.0
Dual value for demand[2]: 1.0
Dual value for demand[3]: 1.0
Dual value for demand[4]: 1.0
Dual value for demand[5]: 1.0
Dual value for demand[6]: 1.0
Dual value for demand[7]: 1.0
Dual value for demand[8]: 1.0
Dual value for demand[9]: 1.0
Dual value for demand[10]: 0.3333333333333333
Dual value for demand[11]: 0.3333333333333333
Dual value for demand[12]: 0.3333333333333333
Dual value for demand[13]: 0.25
Dual value for demand[14]: 0.16666666666666666
Dual value for demand[15]: 0.16666666666666666
Dual value for demand[16]: 0.1111111111111111
Dual value for demand[17]: 0.09090909090909091
Dual value for demand[18]: 0.08333333333333333
Dual value for demand[19]: 0.06666666666666667
Dual value for demand[20]: 0.05263157894736842
```

Hình 11: Giá trị của dual variables

Ví dụ cho yêu cầu từ 1 tới 9, việc tăng yêu cầu lên 1 đơn vị sẽ làm cho hàm mục tiêu tăng 1 đơn vị. Cho những yêu cầu từ 10 tới 12, việc tăng yêu cầu lên 1 đơn vị sẽ làm cho hàm mục tiêu tăng 0.(3) đơn vị.

Hay nói cách khác, việc tăng yêu cầu lên 1 đơn vị cho mảnh i sẽ tốn của ta thêm π_i thanh nguyên liệu. Ngoài ra, ta có thể nói 1 đơn vị tăng ở vế trái (ví dụ do một pattern mới) sẽ tiết kiệm cho ta π_i thanh nguyên liệu.Vì vậy, ta muốn thêm cột mới vào sao cho tối đa hóa khoản tiết kiệm liên quan tới các biến đối ngẫu đồng thời vẫn thỏa điều kiện ràng buộc ban đầu:

$$\max \sum_{i=1}^{I} \pi_i y_i \tag{11}$$

$$\text{s.t.} \sum_{i=1}^{I} w_i y_i \leq W \tag{12}$$

$$y_i \geq 0 \tag{13}$$

$$y_i \in \mathbb{Z} \tag{14}$$

Dây là ví dụ của một **bài toán định giá**(the pricing problem). Nếu bài toán này có giá trị mục tiêu lớn hơn 1 thì ta ước tính việc thêm y làm hệ số của một cột mới sẽ làm giảm giá trị mục tiêu nhiều hơn chi phí của một thanh kim loại thêm:

```julia
function solve_pricing(data::Data, π::Vector{Float64})
    # Get the number of items
    I = length(π)
    # Create a new optimization model using the HiGHS optimizer
    model = Model(HiGHS.Optimizer)
    # Set the model to run silently (without output)
    set_silent(model)
    # Define integer decision variables y[i] for each item, constrained to be non-negative
    @variable(model, y[1:I] >= 0, Int)
    # Add a constraint: the total weight of selected items must not exceed the capacity W
    @constraint(model, sum(data.pieces[i].w * y[i] for i in 1:I) <= data.W)
    # Define the objective function
    @objective(model, Max, sum(π[i] * y[i] for i in 1:I))
    optimize!(model)
    @assert is_solved_and_feasible(model)
    number_of_rolls_saved = objective_value(model)
    # Check if the benefit of the pattern is more than the cost of a new roll plus some tolerance
    if number_of_rolls_saved > 1 + 1e-8
        return SparseArrays.sparse(round.(Int, value.(y)))
    end
    # If the benefit is not sufficient
    return nothing
end
```

Hình 12: Đoạn code giải quyết bài toán định giá

Giờ ta chỉ cần hiện thực giải thuật như đã trình bày ở phần lý thuyết ta sẽ ra được một danh sách các pattern mới:

```
Found new pattern. Total patterns = 21
Found new pattern. Total patterns = 22
Found new pattern. Total patterns = 23
Found new pattern. Total patterns = 24
Found new pattern. Total patterns = 25
Found new pattern. Total patterns = 26
Found new pattern. Total patterns = 27
Found new pattern. Total patterns = 28
Found new pattern. Total patterns = 29
Found new pattern. Total patterns = 30
Found new pattern. Total patterns = 31
Found new pattern. Total patterns = 32
Found new pattern. Total patterns = 33
Found new pattern. Total patterns = 34
Found new pattern. Total patterns = 35
[ Info: No new patterns, terminating the algorithm.
```

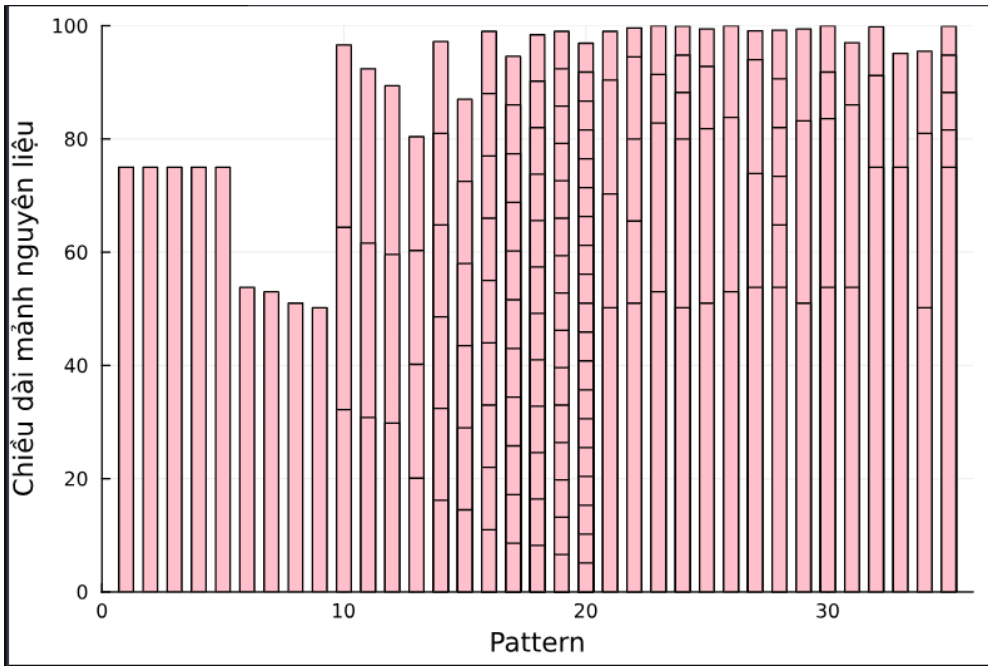
Hình 13: Ta tìm được thêm 15 pattern mới

Ví dụ một pattern mới: pattern[30]:

```
patterns[30] :
  [6 ]  =  1
  [12]  =  1
  [18]  =  2
```

Hình 14: 1 mảnh số 6, 1 mảnh số 12 và 2 mảnh số 18

Ta có đồ thị minh họa trực quan:



Hình 15: Đồ thị minh họa các patterns

Ta có kết quả của bài toán ban đầu như sau:

Row	pattern Int64	rolls Float64
1	1	38.0
2	2	44.0
3	3	30.0
4	21	0.5
5	22	10.2
6	23	14.65
7	24	23.1
8	25	11.25
9	26	21.35
10	28	4.3
11	29	19.55
12	30	11.25
13	31	17.45
14	33	36.0
15	34	11.4
16	35	41.0

Hình 16: Kết quả

Do ngay từ đầu ta đã nới lỏng điều kiện của bài toán thành một bài quy hoạch tuyến tính nên một số dòng có số thanh nguyên liệu ở dạng Float. Ta có thể hành làm tròn lên và được kết quả cuối cùng là cần **341** thanh nguyên liệu.

Hoặc là ta có thể cho lại điều kiện ràng buộc nguyên vào bài toán và tiến hành giải lại. Ta sẽ được bảng kết quả:

Row	pattern	rolls
	Int64	Float64
1	1	38.0
2	2	44.0
3	3	30.0
4	21	1.0
5	22	9.0
6	23	19.0
7	24	19.0
8	25	13.0
9	26	17.0
10	28	2.0
11	29	19.0
12	30	13.0
13	31	18.0
14	33	36.0
15	34	15.0
16	35	41.0

Hình 17: Kết quả sau khi thêm ràng buộc mới

Kết quả mới chỉ còn cần **334** thanh nguyên liệu.

Lưu ý rằng đây có thể chưa phải là giá trị nhỏ nhất trong toàn bộ các nghiệm có thể có của bài toán này bởi vì ta chưa có thêm những cột mới vào trong lúc giải.(Đây là một giải thuật khác tên "Branch and Price"). Dù vậy, giải thuật Column generation đã giúp ta tìm được nghiệm nguyên khả thi hơn cho một bài toán tối ưu hóa vốn không thể giải được!

Kết quả này là đủ hữu ích trong thực tế và sẽ giúp các nhà máy có kế hoạch đúng đắn để giảm thiểu sự lãng phí các thang nguyên liệu.

5.2 Optimal cutting known patterns

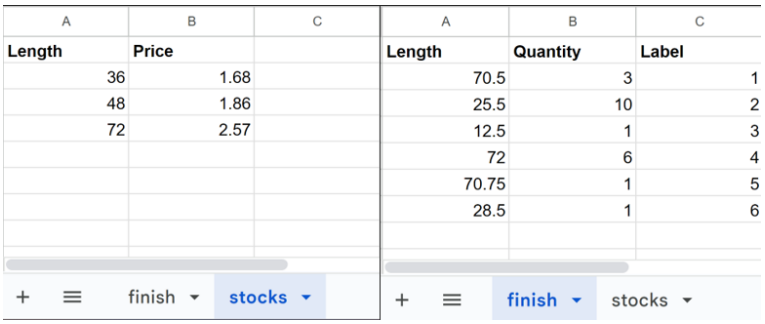
5.2.1 Dẫn nhập

Chúng ta sẽ tạo 2 biến đầu vào là finish và stock được nhập từ file excel.

⋆ Finish là một vector chứa các biến con gồm chiều dài (length), số lượng thanh cần cắt (quantity) và nhãn (label).

⋆ Stock là một vector chứa các biến con gồm chiều dài (length) và giá (price).

Mục tiêu của ta là tìm ra số thanh nguyên liệu ít nhất cần phải cắt sao cho thỏa các yêu cầu đặt hàng.



Length	Price
36	1.68
48	1.86
72	2.57

finish | stocks

Length	Quantity	Label
70.5	3	1
25.5	10	2
12.5	1	3
72	6	4
70.75	1	5
28.5	1	6

finish | stocks

Hình 18: Minh họa INPUT được lấy từ excel

Ta xét một bài toán cụ thể với chiều dài thanh kim loại là $W = 100$ và các yêu cầu còn lại như hình dưới:

```
Finish
 length  quantity  label
  70.50         3  70.50
  25.50        10  25.50
  12.50         1  12.50
  72.00         6  72.00
  70.75         1  70.75
  28.50         1  28.50


Stocks
 length  price
     36   1.68
     48   1.86
     72   2.57
```

Hình 19: Minh họa 6 đơn đặt hàng với độ dài và số lượng tương ứng

5.2.2 Lý thuyết Optimal cutting known patterns

5.2.2.a Problem formulation

Công thức tiêu chuẩn cho vấn đề cắt tấm (1D) (nhưng không phải là duy nhất) bắt đầu với một danh sách gồm m đơn hàng, mỗi đơn hàng yêu cầu n_i tấm có độ dài l_i, $i = 1, \ldots, m$ cần phải cắt từ nguyên liệu thô (thanh) có độ dài L. Chúng ta sau đó xây dựng một danh sách tất cả các kết hợp có thể của các lần cắt (thường được gọi là "mẫu"), liên kết với mỗi mẫu một biến số nguyên dương x_j đại diện cho số lượng các mảnh vật liệu thô sẽ được cắt bằng mẫu j. Bài toán nguyên cắt tấm tuyến tính là:

$$\min \sum_j w_j x_j$$

(1)

$$\sum_j a_{ij} x_j = n_i, \quad \forall i = 1, \ldots, m$$

$$x_j \geq 0 \text{ và nguyên,}$$

trong đó a_{ij} là số lần đơn hàng i xuất hiện trong mẫu j và w_j là chi phí (thường là phần dư $w_j = L - \sum_i a_{ij} l_i$ của mẫu j. Để các phần tử a_{ij}, $i = 1, \ldots, m$ tạo thành một mẫu cắt khả thi, giới hạn sau phải được thỏa mãn:

$$\sum_{i=1}^{m} a_{ij} l_i \leq L,$$

$$a_{ij} \geq 0 \text{ và nguyên.}$$

Thay vì giải quyết vấn đề (1) để giảm thiểu phần dư, chúng ta ưu tiên xem xét vấn đề tương đương để giảm thiểu tổng số lượng thanh được sử dụng, còn được biết đến như là vấn đề đóng gói thùng:

$$\min \sum_j x_j$$

(2)

$$\sum_j a_{ij} x_j = n_i, \quad \forall i = 1, \ldots, m$$

$$x_j \geq 0 \text{ và nguyên.}$$

Nhìn chung, số lượng các mẫu có thể tăng theo hàm mũ theo hàm của m và có thể dễ dàng lên đến hàng triệu. Do đó, việc tạo ra và liệt kê các mẫu cắt có thể trở nên không thực tế.

Một cách tiếp cận để giải quyết vấn đề này là tạo danh sách tất cả các finished parts, danh sách các stocks cho mỗi chiều dài, sau đó sử dụng một tập hợp các biến quyết định nhị phân để gán từng sản phẩm đã hoàn thiện cho một kho cụ thể. Cách tiếp cận này sẽ hiệu quả đối với các vấn đề nhỏ, nhưng độ phức tạp tính toán tăng quá nhanh theo quy mô của vấn đề để có thể áp dụng cho các ứng dụng kinh doanh.

Để giải quyết vấn đề về độ phức tạp của tính toán, vào năm 1961, Gilmore và Gamory đã giới thiệu một cấu trúc dữ liệu bổ sung cho vấn đề mà hiện được gọi là "patterns". Patterns là danh sách finished parts có thể được cắt từ một mặt hàng cụ thể.

Một mẫu p được xác định bởi nguyên liệu s_p, được gán cho mẫu và các số nguyên a_{pf} xác định có bao nhiêu phần thành phẩm loại f được cắt từ nguyên liệu s_p. Một mẫu $p \in P$ là khả thi nếu

$$\sum_{f \in F} a_{pf} l_f^F \leq l_{s_p}$$

Hàm **make_patterns** được định nghĩa bên dưới tạo ra một danh sách một phần các patterns khả thi cho các tập hợp các stocks và các finished parts. Mỗi pattern được biểu diễn dưới dạng từ điển chỉ định một mục stock liên quan và một từ điển các vết cắt chỉ định các bộ phận đã hoàn thành được cắt từ cổ phiếu. Thuật toán này rất đơn giản, nó chỉ xem xét mọi finished parts và các mục stock, sau đó báo cáo số lượng các bộ phận.

Hàm sẽ nhận 2 đối số **stock** và **finish**:

- **stocks**: là một dictionary, trong đó các keys là các định danh của thanh gốc (stock identifiers), và các values là các dictionary con, mỗi dictionary con chứa key 'length' đại diện cho chiều dài của từng thanh gốc.
- **finish**: là một dictionary, trong đó các keys là các định danh của sản phẩm cuối cùng (finish identifiers), và các values là các dictionary con, mỗi dictionary con chứa key 'length' đại diện cho chiều dài yêu cầu của từng sản phẩm cuối cùng.

Hàm **make_patterns** thực hiện các bước sau:

1. Khởi tạo một danh sách rỗng `patterns` để lưu trữ các pattern được tìm thấy.
2. Lặp qua từng sản phẩm cuối cùng f trong `finish`:
   - Khởi tạo biến `feasible` = `False` để theo dõi xem có tìm được pattern nào khả thi cho sản phẩm f hay không.
   - Lặp qua từng thanh gốc s trong `stocks`:
     - Tính số lượng cắt tối đa `num_cuts` bằng cách sử dụng công thức:

$$\texttt{num\_cuts} = \frac{\texttt{length}(s)}{\texttt{length}(f)}$$

     Nếu `num_cuts` > 0:
     - Đặt `feasible` = `True` để đánh dấu rằng đã tìm được pattern khả thi cho sản phẩm f.
     - Tạo một dictionary `cuts_dict` với các khóa là các sản phẩm cuối cùng và các giá trị ban đầu là 0.
     - Đặt `cuts_dict`[f] = `num_cuts` để lưu trữ số lượng cắt của sản phẩm f.
     - Thêm một pattern dictionary vào `patterns`, với `"stock"` là s và `"cuts"` là `cuts_dict`.

     Ngược lại, trả về chuỗi rỗng.
3. Trả về `patterns`.

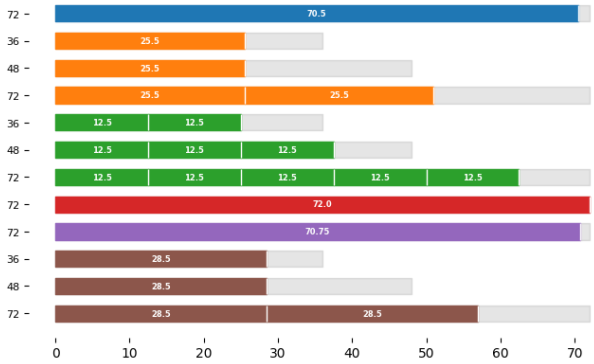
Ở vòng lặp đầu tiên chạy qua từng chiều dài stock/ chiều dài finish(36/70.5) bằng 0 nên lúc này ta sẽ bỏ qua giá trị stock này (cũng như là khi stock bằng 48), tới giá trị cuối cùng là 72 thì chiều dài stock/ chiều dài finish(72/70.5) bằng 1 nên cuts_dict gán bằng num_cuts và bằng 1 với label đầu tiên.

Stock	Cuts					
72	70.5: 1	25.5: 0	12.5: 0	72.0: 0	70.75: 0	28.5: 0

Tương tự ở vòng lặp số 2 chiều dài stock/ chiều dài finish thỏa mãn toàn bộ giá trị của stock và đều lớn hơn 0 nên chiều dài cuts_dict bằng gán giá trị lần lượt là nums_cuts và bằng 1, 1, 2 với label 2.

36	70.5: 0	25.5: 1	12.5: 0	72.0: 0	70.75: 0	28.5: 0
48	70.5: 0	25.5: 1	12.5: 0	72.0: 0	70.75: 0	28.5: 0
72	70.5: 0	25.5: 2	12.5: 0	72.0: 0	70.75: 0	28.5: 0

Tương tự sau khi vòng lặp kết thúc ta thu list của partner như sau:



5.2.2.b Optimal

Sau khi hoàn thành bước tạo Patterns để tối ưu hóa thuật toán thuật toán hơn nữa để tính toán cần cắt bao nhiêu bản sao của mỗi pattern để đáp ứng Quantity về các finished parts với chi phí tối thiểu.

Hãy để chỉ số s_p biểu thị stock được chỉ định bởi pattern p, và hãy để x_{sp} biểu thị số lượng stock s_p được sử dụng.

Đối với một danh sách các pattern nhất định, bài toán tối ưu hóa chi phí tối thiểu là tối ưu hóa tuyến tính số nguyên (a mixed integer linear) hỗn hợp (MILO) tuân theo các ràng buộc về Quantity đối với mỗi mặt hàng đã hoàn thành.

$$\begin{aligned} \min_{p \in P} \quad & \sum_{p \in P} c_{sp} x_{sp} \\ \text{s.t.} \quad & \sum_{p \in P} a_{pf} x_{sp} \geq d_f \quad \forall f \in F \\ & x_{sp} \in \mathbb{Z}_+ \quad \forall p \in P \end{aligned}$$

Để tối ưu thuật toán chúng em dùng mô hình AMPL. Mô hình AMPL được cấu hình để sử dụng bộ giải quyết lập trình tuyến tính hỗn hợp nguyên (Mixed Integer Linear Optimization).

Những lý do nhóm chúng em quyết định dùng mô hình AMPL thay cho cho các mô hình khác là do:

⋆ Tính chất của bài toán: Bài toán tìm số lượng tối ưu của các Patterns là một bài toán tối ưu hóa với biến số là số nguyên, vì chúng ta không thể cắt một phần của một mẫu cắt. Do đó, mô hình MILO(Mixed Integer Linear Optimization)trong AMPL là lựa chọn phù hợp.

⋆ Tính đơn giản của mô hình: Mô hình tối ưu hóa trong bài toán này có cấu trúc tương đối đơn giản, với hàm mục tiêu tuyến tính và các ràng buộc tuyến tính. Mô hình MILO có thể giải quyết loại bài toán tối ưu hóa tuyến tính có biến số nguyên một cách hiệu quả.

⋆ AMPL cung cấp các bộ giải quyết MILO mạnh mẽ như CPLEX, Gurobi hoặc MOSEK. Những bộ giải quyết này có thể giải quyết các bài toán MILO với quy mô lớn một cách hiệu quả, đảm bảo chúng ta có thể tìm được lời giải tối ưu trong thời gian hợp lý.

⋆ Khả năng mở rộng: Nếu như muốn mở rộng bài toán trở nên phức tạp hơn, chẳng hạn như thêm các ràng buộc phi tuyến hoặc yêu cầu về thời gian, mô hình MILO vẫn có thể được mở rộng để xử lý các tình huống như vậy bằng cách sử dụng các tính năng nâng cao của AMPL và các bộ giải quyết MILO.

Trước khi vào định nghĩa, chúng ta phải hiểu mô hình MILO. Tổng quát Minimize:

$$z = c^T * x$$

Với các ràng buộc:

$$A * x \leq b$$

$$x \geq 0$$

$$x_i \text{ integer}, \ i \in I \text{ (I là tập hợp các chỉ số của các biến số nguyên)}$$

Trong đó:

- z: là hàm mục tiêu cần tối ưu, là tổng chi phí cắt các patterns (cost).
- x: là vector các biến quyết định, là số lượng cắt của mỗi patterns.
- c: là vector các chi phí cắt cho mỗi patterns (price).
- A: là ma trận các hệ số ràng buộc, mô tả mối quan hệ giữa các patterns và nhu cầu của các sản phẩm hoàn thiện (finish).
- b: là vector nhu cầu của các sản phẩm hoàn thiện (finish).

5.2.3 Mô tả thuật toán

Hàm `cut_patterns` để giải quyết bài toán tối ưu hóa cắt các patterns từ các thanh nguyên liệu (stocks) để đáp ứng `Quantity` của các sản phẩm hoàn thiện (finish) sử dụng mô hình MILO (Mixed Integer Linear Optimization) trong AMPL.

Hàm này nhận 3 đối số đầu vào: `stocks`, `finish`, `patterns`

- `stocks`, `finish` đối số giống hoàn toàn giống hàm `make_patterns` trên.
- `patterns` là chuỗi danh sách các patterns đã được tạo ra sau khi hàm `make_patterns` thực thi xong.

Dưới đây là chi tiết về các thành phần chính trong hàm:

B1: Khởi tạo mô hình AMPL:

- `m = AMPL()` khởi tạo một đối tượng AMPL mới để xây dựng và giải quyết mô hình tối ưu hóa.

B2: Định nghĩa mô hình trong AMPL:

- Các tập hợp `S`, `F`, `P` được định nghĩa để mô tả các thanh nguyên liệu (`stock`), sản phẩm hoàn thiện (`finish`) và các patterns, tương ứng.
- Các tham số `c`, `a`, `demand_finish` được định nghĩa để mô tả chi phí của các patterns, số lượng các patterns cần thiết cho mỗi sản phẩm hoàn thiện, và nhu cầu của các sản phẩm hoàn thiện.
- Biến quyết định `x` được định nghĩa là số lượng của mỗi patterns cần sử dụng, với điều kiện là số nguyên không âm.
- Hàm mục tiêu `cost` là tổng chi phí của các patterns cần sử dụng, cần được tối thiểu hóa.
- Ràng buộc `demand` đảm bảo rằng tổng số lượng các patterns được sử dụng phải đáp ứng được nhu cầu của các sản phẩm hoàn thiện.

B3: Gán giá trị cho các tập hợp và tham số:

- Các tập hợp `S`, `F`, `P` được gán các giá trị tương ứng từ các đầu vào `stocks`, `finish`, `patterns`.
- Giá trị của các tham số `c`, `a`, `demand_finish` được tính toán và gán từ các đầu vào.

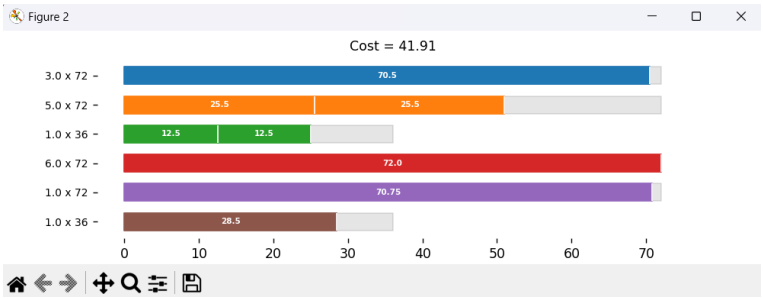
B4: Cấu hình và giải quyết mô hình:

- Chọn bộ giải quyết MILO bằng `m.option["solver"] = SOLVER_MILO`.
- Giải quyết mô hình bằng `m.get_output("solve;")`.

B5: Trả về kết quả:

- Trả về số lượng patterns sử dụng `[m.var["x"][p].value() for p in range(len(patterns))]` và giá trị của hàm mục tiêu `m.obj["cost"].value()`.

Sau khi thuật toán chạy, danh sách patterns mới và patterns tối ưu nhất sẽ được tạo ra cả về số lượng lẫn chi phí cắt.



Hình 20: Mô tả của hình ảnh

6 Phân tích

6.1 Column Generation

Ta tiến hành dùng **@time** macro của Julia để đo thời gian thực thi của phần hiện thực giải thuật này:

```
[ Info: No new patterns, terminating the algorithm.
                                  Time                    Allocations
      Tot / % measured:         592ms /  93.5%           51.2MiB /  99.2%

Section           ncalls     time    %tot     avg     alloc    %tot      avg
Total Execution        1    554ms  100.0%   554ms   50.8MiB  100.0%  50.8MiB
```

Hình 21: Thời gian chạy giải thuật Column Generation

Tổng thời gian thực thi

- **Time Analysis:** 592 milliseconds (ms)
  - Đây là tổng thời gian thực thi của đoạn code hiện thực giải thuật Column eneration
  - Thời gian thực thi này bao gồm tất cả các hoạt động như: tiền xử lý dữ liêu, setup model, tối ưu hóa và hậu xử lý dữ liệu.

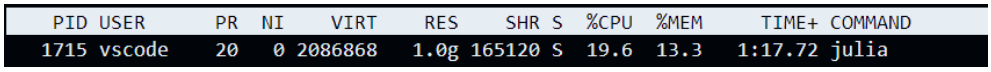
Tổng bộ nhớ được cấp phát

- **Phân tích bộ nhớ cấp phát:** 51.2 Megabytes (MiB)
  - Đây là khối bộ nhớ được cấp phát trong quá trình thực thi đoạn code trên
  - Cấp phát bộ nhớ dùng cho biến, cho các cấu trúc dữ liệu và các phép tính tức thời.

Nhận xét

- **Thời gian thực thi:**
  - Tổng thời gian thực thi là 592 ms nghĩa là giải thuật chạy khá nhanh - một dấu hiệu tốt cho performance
  - Tuy nhiên việc này phụ thuộc vào use case trong từng trường hợp cụ thể. Nếu ta cần thời gian thực thi nhanh hơn nữa, ta phải optimize những đoạn code cụ thể
- **Lưu lượng bộ nhớ sử dụng:**
  - Tổng bộ nhớ dùng là 51.2 MiB nghĩa là giải thuật dùng một lượng vừa phải.
  - Tuy nhiên nếu ta cần quan tâm đến vấn đề về dung lượng bộ nhớ (chạy trong các môi trường mà bộ nhớ bị giới hạn vd: vi điều khiển) thì ta cần phải tối ưu bộ nhớ bằng việc thay đổi các cấu trúc dữ liệu được dùng trong giải thuật.

Một điểm tham chiếu khác ta có thể dùng để đánh giá tốc độ của giải thuật là dùng command **top** (ở lúc đo là trong Ubuntu) để quan sát CPU và Memory:



Hình 22: CPU and Memory Usage khi chạy câu lệnh top

Quá trình thực thi giải thuật chiếm 19.6% của CPU, một lượng khá. Điều này có nghĩa là việc thực hiện giải thuật này bằng CPU tiêu tốn khá nhiều tài nguyên của CPU cho việc tính toán. Quá trình cần 13.3% tổng lưu lượng bộ nhớ khả dụng. Đây là một con số lớn và có thể được tối ưu hơn bằng cách lựa chọn các cấu trúc dữ liệu hợp lý cũng như giảm thiểu những tính toán yêu cầu tài nguyên lớn.

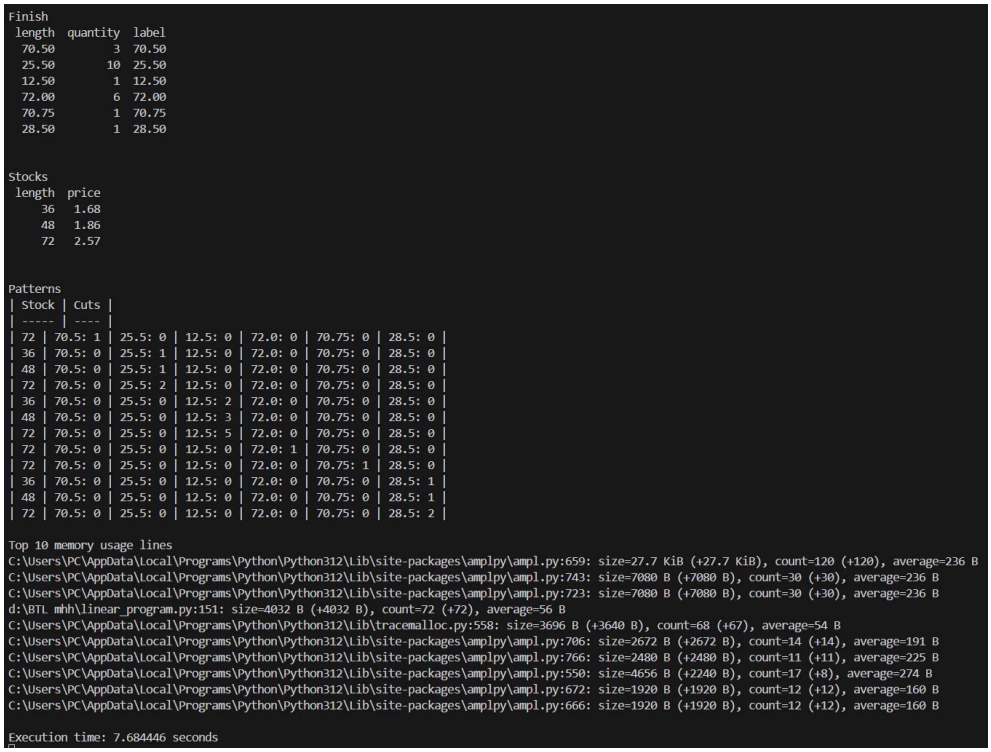
6.2 Optimal cutting known patterns

Độ phức tạp của Optimal cutting known patterns - Algorithm được tính toán như sau: Độ phức tạp thời gian (Time Complexity): O(m * n) cho việc tạo mẫu cắt, và O(p * (m + n)) cho việc thiết lập mô hình lập trình toán học.

Trong đó:

- p là số lượng mẫu cắt khả thi (`patterns`) được tạo ra bởi hàm make_patterns.
- m là số lượng yêu cầu hoàn thiện (`finish`) trong hàm cut_patterns mà bạn cần đáp ứng.
- n là số lượng thanh nguyên liệu (`stocks`) mà bạn có để cắt.

Ta tiến hành dùng thư viện "time" và "tracemalloc" của python để đo thời gian thực thi và memory được sử dụng trong giải thuật này:



Hình 23: Thời gian thực thi và memory giải thuật Optimal cutting known patterns

Tổng thời gian thực thi

- **Time Analysis:** 7684.446 milliseconds (ms)
  - Đây là tổng thời gian thực thi của đoạn code hiện thực giải thuật Optimal cutting known patterns
  - Thời gian thực thi này bao gồm tất cả các hoạt động như: make_patterns, cut_patterns, tối ưu hóa và hậu xử lý dữ liệu.
  - với m, n, p trong ví dụ trên lần lượt là 6, 3 và 12.

Tổng bộ nhớ được cấp phát

- **Phân tích bộ nhớ cấp phát:** 27.7 KiB + 7080 B + 7080 B + 4032 B + 3640 B + 2672 B + 2480 B + 4656 B + 1920 B + 1920 B = 62,159 B, tương đương với khoảng 60.71 KiB.
  - Đây là khối bộ nhớ được cấp phát trong quá trình thực thi đoạn code trên
  - Cấp phát bộ nhớ dùng cho biến, cho các cấu trúc dữ liệu và các phép tính tức thời.

Nhận xét:

- **Độ phức tạp của thuật toán:**
  - Tạo mẫu cắt (make_patterns): O(m * n)
  - Đây là độ phức tạp tuyến tính, tỷ lệ thuận với số lượng yêu cầu hoàn thiện (m) và số lượng kho (n). Điều này cho thấy thuật toán sẽ chạy nhanh khi số lượng yêu cầu và kho không quá lớn.
  - Thiết lập mô hình lập trình toán học: O(p * (m + n))
  - Độ phức tạp ở đây phụ thuộc vào số lượng mẫu cắt khả thi (p) và tổng số lượng yêu cầu hoàn thiện (m) và số lượng kho (n). Điều này cho thấy khi số lượng mẫu cắt khả thi tăng lên, độ phức tạp sẽ tăng lên tương ứng.
  - Ta có thể thấy thuật toán này có thể được sử dụng hiệu quả trong các bài toán cắt tối ưu với số lượng yêu cầu, thanh nguyên liệu và mẫu cắt không quá lớn.
  - Khi số lượng các thông số này tăng lên, độ phức tạp của thuật toán cũng tăng lên, do đó cần cân nhắc sử dụng các thuật toán tối ưu hơn hoặc áp dụng các kỹ thuật tối ưu hóa khác.
  - Tóm lại, độ phức tạp của thuật toán Optimal cutting known patterns - Algorithm là tuyến tính, cho phép sử dụng hiệu quả với các bài toán có quy mô vừa phải.
- **Thời gian thực thi:**
  - Thời gian thực thi khá dài: 7684.446 milliseconds (ms) tương đương khoảng 7.7 giây.
  - Điều này có thể cho thấy khi số lượng yêu cầu hoàn thiện (m = 6), số lượng kho (n = 3) và số lượng mẫu cắt khả thi (p = 12) không quá lớn, thuật toán vẫn cần khá nhiều thời gian để thực hiện các hoạt động như make_patterns, cut_patterns, tối ưu hóa và hậu xử lý dữ liệu.
  - Như đã phân tích ở trước, độ phức tạp thời gian của thuật toán này là O(m * n + p * (m + n)).
  - Với m = 6, n = 3 và p = 12, ta có thể ước tính độ phức tạp thời gian là khoảng O(6 * 3 + 12 * (6 + 3)) = O(18 + 108) = O(126).
  - Điều này phù hợp với thời gian thực thi khá dài (7.7 giây) vì độ phức tạp tăng khi các thông số đầu vào tăng lên.
  - Tóm lại, thời gian thực thi 7.7 giây cho một bài toán với quy mô vừa phải cho thấy thuật toán vẫn còn chậm và cần được tối ưu hóa để có thể áp dụng hiệu quả trong thực tế, đặc biệt khi quy mô bài toán tăng lên.
- **Lưu lượng bộ nhớ sử dụng:**
  - Tổng lượng bộ nhớ được cấp phát là 62,159 B, tương đương với khoảng 60.71 KiB. Với một bài toán có quy mô vừa phải (m = 6, n = 3, p = 12), việc sử dụng 60.71 KiB bộ nhớ là khá lớn.
  - Với lượng bộ nhớ sử dụng lớn như vậy, nên xem xét các cách tối ưu hóa để giảm lượng bộ nhớ cần thiết.
  - Việc giảm lượng bộ nhớ sử dụng sẽ giúp cải thiện hiệu suất và khả năng mở rộng của thuật toán, đặc biệt khi quy mô bài toán tăng lên.

6.3 So sánh 2 giải thuật Column Generation và Optimal cutting known patterns

- **Về thời gian thực thi:**
  - Thuật toán Column Generation có thời gian thực thi nhanh hơn nhiều so với Optimal Cutting Known Patterns (592ms vs 7684.446ms).
  - Điều này cho thấy Column Generation là một thuật toán tối ưu hơn về mặt thời gian thực thi.
- **Về bộ nhớ được cấp phát:**
  - Thuật toán Column Generation sử dụng nhiều bộ nhớ hơn so với Optimal Cutting Known Patterns (51.2MB vs 60.71KB).
  - Điều này cho thấy Column Generation có nhu cầu bộ nhớ lớn hơn, có thể do các cấu trúc dữ liệu và phép tính phức tạp hơn.
- **Tổng kết lại:** Với thời gian thực thi nhanh hơn và bộ nhớ sử dụng nhiều hơn, thuật toán Column Generation có vẻ tối ưu hơn so với Optimal Cutting Known Patterns, đặc biệt là khi tập dữ liệu lớn và yêu cầu về thời gian thực thi quan trọng hơn.Tuy nhiên khi hệ thống của bạn không đủ mạnh và bộ nhớ ít thì thuật toán Optimal Cutting Known Patterns có thể sẽ tối ưu hơn.

7 Kết luận và hướng đi tiếp theo

Thông qua bài tập lớn, nhóm chúng em đã tìm hiểu và hiện thực 2 giải thuật cơ bản trong bài toán Cutting-stock, phân tích và so sánh độ phức tạp chúng. Cụ thể, chúng em đã:

- Mô tả chi tiết và cài đặt: Hai giải thuật được chọn là (Column Generation và Optimal cutting known patterns). Chúng em đã tiến hành mô tả chi tiết các bước thực hiện và hiện thực hoá chúng bằng ngôn ngữ lập trình phù hợp.
- Phân tích độ phức tạp: Chúng em đã phân tích độ phức tạp thời gian và không gian của từng giải thuật, đưa ra so sánh cụ thể về ưu và nhược điểm của mỗi giải thuật trong các trường hợp khác nhau của bài toán Cutting-stock.
- Thực nghiệm và đánh giá: Qua việc thực hiện các thử nghiệm trên các bộ dữ liệu thực tế, chúng em đã thu thập kết quả và đưa ra đánh giá về hiệu suất của từng giải thuật. Kết quả cho thấy mỗi giải thuật có hiệu quả khác nhau tuỳ thuộc vào kích thước và cấu trúc của bộ dữ liệu đầu vào.

Để nâng cao chất lượng và hiệu quả của nghiên cứu, nhóm chúng em dự định:

- Cải tiến giải thuật: Tìm kiếm và nghiên cứu các phương pháp tối ưu hóa, cải tiến hai giải thuật hiện tại nhằm giảm thiểu độ phức tạp và tăng cường hiệu suất.
- Đánh giá và tối ưu hóa hiệu suất: Thực hiện các bài kiểm tra và đánh giá hiệu suất của các giải thuật trên các hệ thống và nền tảng khác nhau, từ đó đề xuất các phương án tối ưu hóa cho từng môi trường cụ thể.

Nhóm chúng em tin rằng, với các hướng đi trên, nghiên cứu về giải thuật cho bài toán Cutting-stock sẽ được mở rộng và nâng cao, đem lại nhiều giá trị thiết thực cho lĩnh vực công nghiệp và những lĩnh vực khác trong thực tế.

8 Thông tin các file trong bài nộp

Bài nộp gồm đầy đủ các file sau:

- **Assignment-CO2011-CSE233-2210284-Report.pdf**: báo cáo viết bằng tiếng Việt.
- Folder **Assignment-CO2011-CSE233-2210284-LatexFiles**: thư mục chứa file .tex và các files liên quan.
- **Assignment-CO2011-CSE233-2210284-ColumnGeneration.jl**: mã nguồn giải thuật 1 viết bằng Julia.
- **Assignment-CO2011-CSE233-2210284-inputBTL.xlsx**: Dữ liệu cho giải thuật 2
- **Assignment-CO2011-CSE233-2210284-OptimalCutting.py**: mã nguồn giải thuật 2 viết bằng Python.
- **Assignment-CO2011-CSE233-2210284-ReadMe.md**: hướng dẫn để chạy source code giải thuật 1 và 2.
- **Assignment-CO2011-CSE233-2210284-MeetingMinutes.pdf**: biên bản ghi lại các buổi họp, thảo luận của nhóm trong quá trình làm bài.

Tài liệu tham khảo

[1] Alain Chabrier. *Column Generation techniques*. https://medium.com/@AlainChabrier/column-generation-techniques-6a414d723a64. 2019.

[2] Wikipedia contributors. *Cutting stock problem*. https://en.wikipedia.org/wiki/Cutting_stock_problem. 2023.

[3] *Dantzig-Wolfe decomposition*. http://encyclopediaofmath.org/index.php?title=Dantzig-Wolfe_decomposition&oldid=50750. Encyclopedia of Mathematics.

[4] Jacques Desrosiers and Marco Lübbecke. "A Primer in Column Generation". In: *Column Generation*. Springer, 2006, pp. 7–14. DOI: 10.1007/0-387-25486-2_1.

[5] Gerard. *Personnel and Vehicle Scheduling, Column Generation*. Slide 12, https://slideplayer.com/slide/6574/. 2005.

[6] David S Johnson. "The NP-completeness column: an ongoing guide". In: *Journal of algorithms* 6.3 (1985), pp. 434–451.

[7] Krzysztof Postek - Alessandro Zocca - Joaquim Gromicho - Jeffrey Kantor1. *Extra Material: Cutting Stock*. https://ampl.com/mo-book/notebooks/05/cutting-stock.html#optimal-cutting-using-known-patterns. 2023.

[8] Hans Kellerer et al. "Introduction to NP-Completeness of knapsack problems". In: *Knapsack problems* (2004), pp. 483–493.

[9] L.A. Wolsey. *Integer Programming*. Column Generation Algorithms. Wiley, 1998, pp. 185–189.