TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Báo cáo code thuật toán đơn hình

Thành viên:

1. Võ Tuấn Kiệt 20280056

I. Thông tin nhóm :

STT	MSSV	Tên	Email	Nhóm trưởng	Nhiệm vụ thực hiện	Đánh giá
1	20280056	Võ Tuấn Kiệt	20280056@student.hcmus.edu.vn	X	Làm tất cả	Hoàn thành nhiệm vụ

II. Thông tin bài làm

1. Những trường hợp xử lý được :

- Biến tùy ý
- Ràng buộc biến tùy ý (<=, >=, =)
- Ràng buộc tùy ý (<=, >= , =)
- Hàm mục tiêu dạng min / max
- Trường hợp bài toán đơn hình bình thường Danzig
- Trường hợp bài toán đơn hình 2 pha (có b âm)
- Trường hợp bài toán đơn hình Bland (trường hợp Danzig xuất hiện vòng lặp)
- Giao diện người dùng

2. Những trường hợp chưa xử lý được :

- Giao diện người dùng chưa tối ưu
- Chưa xử lý được các lỗi nếu người dùng nhập sai định dạng yêu cầu (ví dụ yêu cầu nhập số biến nhưng người dùng nhập chữ ,…)

III. Hướng dẫn code :

1. Phần mềm sử dụng :

Python 3.11.3

Các thư viện được sử dụng trong python :

numpy = = 1.24.3

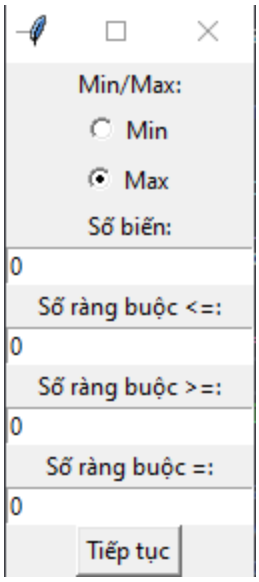
tk = = 0.1.0

Hướng dẫn các bước thực hiện :

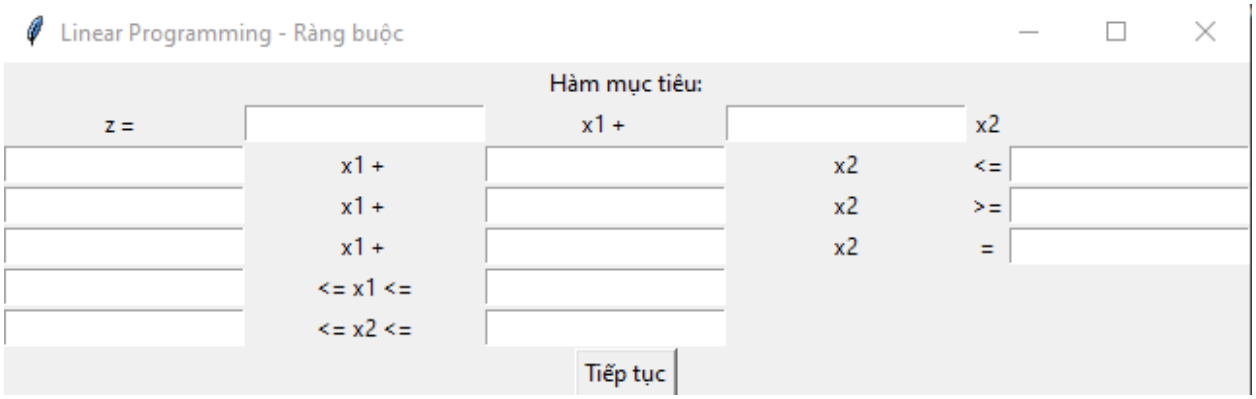
- Cài đặt python trên trang chủ của python
- Mở terminal và hướng đường dẫn đến thư mục chứa file code
- Sử dụng câu lệnh “pip install -r requirements.txt”
- Sau đó sử dụng câu lệnh “python main.py”

2. Hướng dẫn nhập dữ liệu :

Sau khi chạy các câu lệnh bên trên, cửa sổ người dùng hiện ra để nhập dữ liệu như sau :



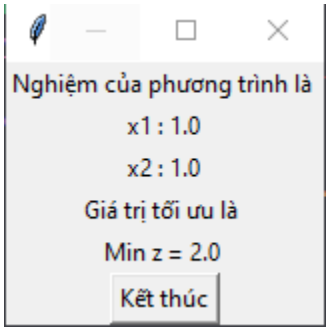
Nếu bài toán là max thì chọn là max, ngược lại chọn là min
Nhập số biến , phải là một số nguyên > 0
Nhấp số ràng buộc (<= , >=, =), phải là một số nguyên >= 0
Sau đó nhấn nút tiếp tục thì cửa sổ sau hiện ra :



Lần lượt nhập các hệ số cho hàm mục tiêu, các ràng buộc, là một số nguyên , số thực (ví dụ 0.33) hoặc phân số (ví dụ 1/3). Sau khi nhập xong thì bấm nút tiếp tục.

Nhập các ràng buộc biến , là số nguyên, số thực hoặc phân số, nếu biến không có giới hạn thì nhập là None, ví dụ x1 >= 0 thì nhập là 0, None hoặc nếu x1 không có giới hạn gì thì nhập là None, None

Sau đó thì cửa sổ kết quả sẽ hiện ra cho biết nghiệm tối ưu và giá trị tối ưu, nhấn nút kết thúc để kết thúc.



Lưu ý là phải nhập đúng định dạng ở trên để chương trình không bị lỗi ngoài ý muốn. Nếu có lỗi xảy ra thì bên trong terminal bấm Ctrl + C, sau đó chạy lại câu lệnh "python main.py" để nhập lại bài toán. Trường hợp bấm Ctr + C không được thì tắt terminal rồi bật lại rồi làm lại tương tự như các bước ở trên.

3. Giải thích code

Trong thư mục có 3 file code là App.py, Simplex.py và main.py.

- App.py là file code để hiển thị giao diện người dùng cho phép nhập và xuất kết quả ra màn hình, do đó em sẽ không giải thích code phần này vì nó không phải trọng tâm.
- Simplex.py là file code chạy thuật toán chính để giải bài toán quy hoạch tuyến tính, do đó em chỉ tập trung vào giải thích code của file này.
- main.py là file code để chạy chương trình, không có gì để giải thích.

Giải thích code trong file Simplex.py :

a) class SimplexMethod và hàm khởi tạo `__init__` :

Đầu tiên khai báo một lớp có tên là "SimplexMethod" và khởi tạo một đối tượng của lớp đó bằng hàm __init__ . Ý nghĩa của các tham số và thuộc tính trong hàm:

1. `c`: Là một mảng 1 chiều chứa hệ số của các biến trong hàm mục tiêu của bài toán quy hoạch tuyến tính.

2. `A_ub`: Là một mảng 2 chiều chứa hệ số của các biến trong các ràng buộc tuyến tính dạng ">= ", có thể None nếu không có ràng buộc >=.

3. `b_ub`: Là một mảng 1 chiều chứa giá trị bên phải của các ràng buộc tuyến tính dạng ">= ", có thể None nếu không có ràng buộc >=.

4. `A_lb`: Là một mảng 2 chiều chứa hệ số của các biến trong các ràng buộc tuyến tính dạng "<= ", có thể None nếu không có ràng buộc <=.

5. `b_lb`: Là một mảng 1 chiều chứa giá trị bên phải của các ràng buộc tuyến tính dạng "<= ", có thể None nếu không có ràng buộc <=.

6. `A_eq`: Là mảng 2 chiều chứa hệ số của các biến trong các ràng buộc tuyến tính dạng "=", có thể None nếu không có ràng buộc =.

7. `b_eq`: Là một mảng 1 chiều chứa giá trị bên phải của các ràng buộc tuyến tính dạng "=", có thể None nếu không có ràng buộc >=.

8. `bounds`: Là một mảng 2 chiều chứa ràng buộc của các biến, có thể là None nếu không có ràng buộc.

9. `prob`: Là biến số nguyên có giá trị là 1 nếu bài toán là min hoặc 0 nếu bài là max.

Tất cả các mảng 1 chiều và 2 chiều ở bên trên đều được lưu trữ dưới dạng danh sách (list)

Hàm `__init__` tạo một named tuple `_LPProblem` với các trường tương ứng với các tham số truyền vào, sau đó khởi tạo thuộc tính `lp` là một đối tượng `_LPProblem` với các giá trị tương ứng được truyền vào. Điều này giúp lưu trữ các thông tin cần thiết để giải quyết bài toán quy hoạch tuyến tính bằng phương pháp đơn hình.

b) **Hàm `_clean_inputs` :**
Hàm `_clean_inputs` dùng chuẩn hóa dữ liệu người dùng nhập vào để có thể giải bài toán quy hoạch tuyến tính

Quá trình xử lý chi tiết của hàm `_clean_inputs`:

1. Trích xuất các giá trị từ named tuple `self.lp` và gán chúng cho các biến tương ứng (c, A_ub, b_ub, A_lb, b_lb, A_eq, b_eq, bounds, prob).

2. Kiểm tra xem vector hệ số của hàm mục tiêu (c) có tồn tại hay không. Nếu không tồn tại, xuất ra lỗi ValueError.

3. Xác định số chiều của vector hệ số (n_x) và chuyển đổi nó thành một mảng numpy có kiểu dữ liệu float64. Nếu kích thước của mảng c là 1, thì chuyển nó thành một vector hàng (1D).

4. Kiểm tra xem ma trận hệ số của ràng buộc ">= " (A_ub) có tồn tại hay không. Nếu không tồn tại, tạo một ma trận zero có kích thước (0, n_x).

5. Chuyển đổi ma trận A_ub thành một mảng numpy có kiểu dữ liệu float.

6. Kiểm tra xem vector bên phải của ràng buộc ">= " (b_ub) có tồn tại hay không. Nếu không tồn tại, tạo một vector rỗng.

7. Chuyển đổi vector b_ub thành một mảng numpy có kiểu dữ liệu float và loại bỏ các chiều dư thừa. Nếu kích thước của mảng b_ub là 1, chuyển nó thành một vector hàng (1D).

8. Tương tự như bước 4 và 5, kiểm tra và xử lý ma trận hệ số và vector bên phải của ràng buộc "<=" (A_lb và b_lb).

9. Tương tự như bước 4 và 5, kiểm tra và xử lý ma trận hệ số và vector bên phải của ràng buộc "=" (A_eq và b_eq).

10. Chuyển đổi danh sách bounds thành một mảng numpy có kiểu dữ liệu float.

11. Xử lý các giá trị None trong mảng bounds_clean. Các giá trị None trong cột thứ nhất được thay thế bằng -np.inf và trong cột thứ hai được thay thế bằng np.inf.

12. Gán lại các giá trị đã xử lý sạch vào named tuple `_LPProblem`.

Cuối cùng, thuộc tính `self.lp` được gán giá trị mới là một named tuple `_LPProblem` chứa các giá trị đã được chuẩn hóa.

c) Hàm `_get_Abc`:

Hàm `_get_Abc` được sử dụng để chuyển đổi bài toán quy hoạch tuyến tính về dạng chính tắc (canonical form) để sử dụng phương pháp đơn hình.

Cụ thể hàm `_get_Abc` chuyển đổi bài toán dạng :

Minimize/ Maximize::

c @ x

Subject to:

A_ub @ x <= b_ub

A_lb @ x >= b_lb

A_eq @ x = b_eq

lb <= x <= ub

Thành :

Minimize::

c @ x

Subject to::

A @ x = b

x >= 0

Dưới đây là quá trình xử lý chi tiết của hàm `_get_Abc`:

1. Trích xuất các giá trị từ named tuple `self.lp` và gán chúng cho các biến tương ứng (c, A_ub, b_ub, A_lb, b_lb, A_eq, b_eq, bounds, prob).

2. Kiểm tra giá trị của `prob`. Nếu `prob` là 0 tức là bài toán max, đảo dấu vector hệ số của hàm mục tiêu (c) và cập nhật named tuple `self.lp` với các giá trị mới.

3. Ghép ma trận hệ số của ràng buộc ">= " (A_ub) và ràng buộc "<=" (-A_lb) thành một ma trận mới (A_ub) bằng cách sử dụng hàm `np.concatenate`.

4. Ghép vector bên phải của ràng buộc ">= " (b_ub) và ràng buộc "<=" (-b_lb) thành một vector mới (b_ub) bằng cách sử dụng hàm `np.concatenate`.

5. Sao chép mảng `bounds` để tạo một bản sao mới (bounds) để tránh thay đổi mảng gốc.

6. Xử lý ràng buộc của biến sao cho tất cả các biến chỉ có giới hạn không âm (>=). Chia nhỏ quá trình này thành các bước nhỏ:

- Lấy danh sách giới hạn dưới (lbs) và giới hạn trên (ubs).

- Xác định các trường hợp: không giới hạn dưới (lb_none), không giới hạn trên (ub_none), và có giới hạn dưới (lb_some), có giới hạn trên (ub_some).

- Xử lý trường hợp không có giới hạn dưới và có giới hạn trên cùng một biến (l_nolb_someub) (Ví dụ trường hợp -inf <= xi <= ub). Đảo dấu các giới hạn trên và dưới, đảo dấu hệ số tương ứng trong vector hệ số của hàm mục tiêu (c) và trong ma trận hệ số của ràng buộc (A_ub, A_eq) nếu có. Cụ thể là thay xi=-xi để biến -inf <= xi <= ub thành -ub <= xi <= inf

- Xử lý trường hợp có giới hạn dưới và giới hạn trên (lb <= xi <= ub) : tách lb <=xi <=ub thành lb <= xi và thêm xi <= ub vào A_ub. Cập nhật A_ub và b_ub tương ứng.

Sau bước này , tất cả các ràng buộc đều có dạng lb <= xi <= inf (chỉ có giới hạn dưới) hoặc -inf <= xi <= inf (không có giới hạn)

7. Ghép ma trận A_ub và A_eq theo chiều dọc để tạo ma trận A1. Ghép vector bên phải của ràng buộc ">= " (b_ub) và ràng buộc "=" (b_eq) theo chiều dọc để tạo vector b.

8. Ghép vector hệ số của hàm mục tiêu (c) và một vector chứa các phần tử 0 với kích thước bằng số hàng của ma trận A_ub để tạo vector c mới (tương ứng với số biến phụ được thêm vào A_ub để từ <= thành =).

9. Xử lý biến tự do. Chia nhỏ quá trình này thành các bước nhỏ:

- Xác định biến tự do (l_free) và chỉ mục của chúng (i_free).

- Tạo các biến mới (x+) và (x-) bằng cách chia đôi các biến tự do.

- Ghép biến mới vào vector hệ số của hàm mục tiêu (c) và ma trận hệ số của ràng buộc (A1).

10. Thêm biến phụ (slack variables) để biến các ràng buộc "<=" thành "=" bằng cách thêm ma trận đơn vị vào ma trận A1. Cập nhật ma trận A và vector b tương ứng.

11. Xử lý giới hạn dưới: biến đổi các giới hạn dưới thành 0, cụ thể : lb <= xi => 0<=xi-lb => thay xi bởi xi + lb. Cập nhật vector hệ số của hàm mục tiêu (c) và vector b tương ứng.

12. Trả về các giá trị A, b, c và c0 (hệ số tự do) như kết quả.

Hàm `_get_Abc` được sử dụng để chuẩn hóa bài toán tối ưu tuyến tính thành dạng chính tắc phù hợp để áp dụng phương pháp đơn hình.

Trước khi đi vào thuật toán chính , em sẽ trình bày ý tưởng giải của bài toán quy hoạch tuyến tính trước, để có thể dễ đối chiếu với phần code.

Các bước giải bài toán quy hoạch, được trích dẫn từ chương 5 : The simplex method trong sách Linear Programming and Extensions. (1 pp. 94-119) (1)

Outline of the Procedure.

A. Arrange the original system of equations so that all constant terms b_i are positive or zero by changing, where necessary, the signs on both sides of any of the equations.

B. Augment the system to include a basic set of *artificial* or *error variables* $x_{N+1} \geq 0,\ x_{N+2} \geq 0, \ldots, x_{N+M} \geq 0$, so that it becomes

(4)

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N + x_{N+1} &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N + x_{N+2} &= b_2 \ (b_i \geq 0) \\
\cdot \quad \cdot \quad \cdot & \quad \cdot \\
\cdot \quad \cdot \quad \cdot & \quad \cdot \\
\cdot \quad \cdot \quad \cdot & \quad \cdot \\
a_{M1}x_1 + a_{M2}x_2 + \ldots + a_{MN}x_N + x_{N+M} &= b_M \\
c_1x_1 + c_2x_2 + \ldots + c_Nx_N + (-z) &= 0
\end{aligned}$$

and

(5) $$x_j \geq 0 \quad (j = 1, 2, \ldots, N, N+1, \ldots, N+M)$$

C. (Phase I): Use the simplex algorithm (with no sign restriction on z) to find a solution to (4) and (5) which minimizes the sum of the artificial variables, denoted by w:

(6) $$x_{N+1} + x_{N+2} + \ldots + x_{N+M} = w$$

Equation (6) is called the *infeasibility form*. The initial feasible canonical system for Phase I is obtained by selecting as basic variables $x_{N+1}, x_{N+2}, \ldots, x_{N+M}, (-z), (-w)$ and eliminating these variables (except w) from the w form by subtracting the sum of the first M equations of (4) from (6), yielding

(7)

Admissible Variables	Artificial Variables		
$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N$	$+x_{N+1}$		$= b_1$
$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N$	$+x_{N+2}$		$= b_2$
. . .	.		.
. . .	.		.
. . .	.		.
$a_{M1}x_1 + a_{M2}x_2 + \ldots + a_{MN}x_N$	$+ x_{N+M}$		$= b_M$
$c_1x_1 + c_2x_2 + \ldots + c_Nx_N$		$-z$	$= 0$
$d_1x_1 + d_2x_2 + \ldots + d_Nx_N$		$-w$	$= -w_0$

where $b_i \geq 0$ and

$$d_j = -(a_{1j} + a_{2j} + \ldots + a_{Mj}) \quad (j = 1, 2, \ldots, N) \tag{8}$$
$$-w_0 = -(b_1 + b_2 + \ldots + b_M)$$

Writing (7) in detached coefficient form constitutes the *initial tableau* for Phase I (see Table 5-2-I).

D. If Min $w > 0$, then no feasible solution exists and the procedure is terminated. On the other hand, if Min $w = 0$, initiate Phase II of the simplex algorithm by (i) dropping from further consideration all non-basic variables x_j whose corresponding coefficients d_j are positive (not zero) in the final modified w-equation; (ii) replacing the linear form w (as modified by various eliminations) by the linear form z, after first eliminating from the z-form all basic variables. (In practical computational work the elimination of the basic variables from the z-form is usually done on each iteration of Phase I; see Tables 5-2-I, 5-2-II, and 5-2-III. If this is the case, then the modified z-form may be used immediately to initiate Phase II.)

E. (Phase II): Apply the simplex algorithm to the adjusted feasible canonical form at end of Phase I to obtain a solution which minimizes the value of z or a class of solutions such that $z \to -\infty$.

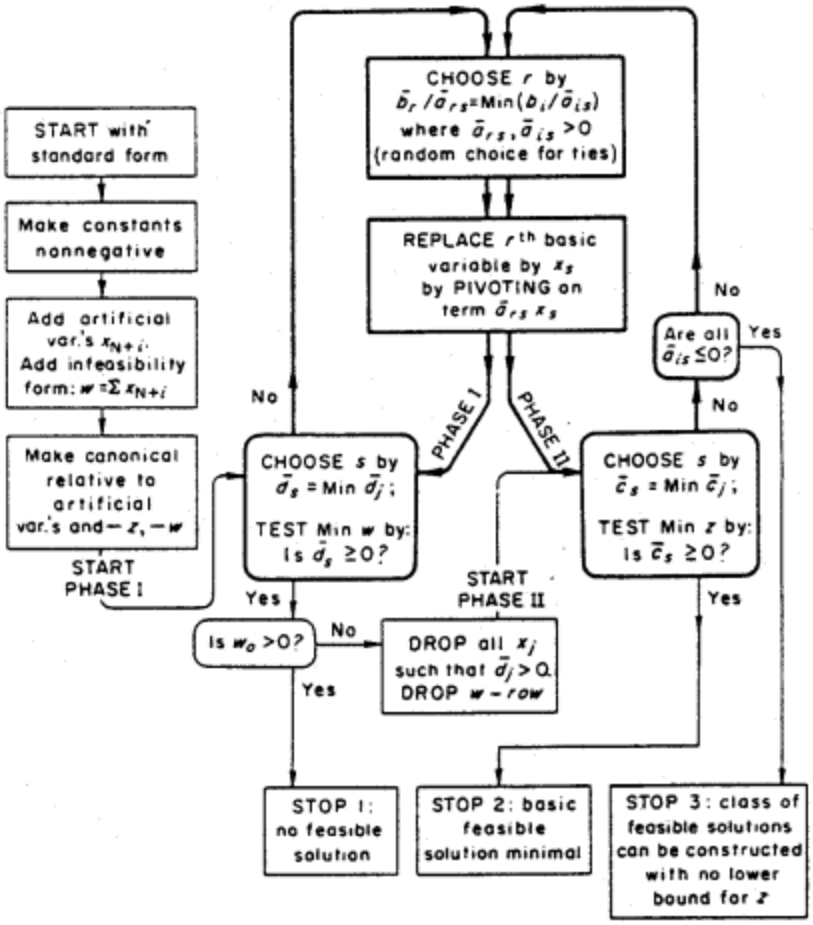


Figure 5-2-I. Flow diagram of the simplex method.

Thực ra phương pháp giải của bài toán này cũng hoàn toàn tương tự như cách giải bằng phương pháp đơn hình 2 pha từ vựng, chỉ là cách xây dựng khác đi một chút để phù hợp với việc tổng quát hóa bài toán, cũng như để cho việc chuyển từ pha 1 qua pha 2 trong lúc code được dễ dàng hơn (không phải handle nhiều trường hợp nhỏ khác nhau dẫn đến khó khăn trong lúc lập trình).

d) Hàm `_pivot_col` :

Hàm `_pivot_col` được sử dụng để xác định cột của biến để vào cơ sở . Quá trình xử lý chi tiết của hàm `_pivot_col`:

1. Nhận đầu vào là ma trận `T` của bài toán quy hoạch tuyến tính. Ma trận `T` có các dòng thể hiện các ràng buộc và hàm mục tiêu, và các cột thể hiện các biến và hệ số tương ứng.

Cụ thể ma trận T có dạng sau :

Admissible Variables	Artificial Variables	
$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N$	$+x_{N+1}$	$= b_1$
$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N$	$+x_{N+2}$	$= b_2$
. . .	.	.
. . .	.	.
. . .	.	.
$a_{M1}x_1 + a_{M2}x_2 + \ldots + a_{MN}x_N$	$+ x_{N+M}$	$= b_M$
$c_1x_1 + c_2x_2 + \ldots + c_Nx_N$		$-z = 0$
$d_1x_1 + d_2x_2 + \ldots + d_Nx_N$		$-w = -w_0$

2. Tạo một ma trận giả mặc định (masked array) `ma` từ hàng cuối cùng của ma trận `T`, nghĩa là vector hệ số của hàm mục tiêu (w với pha 1, z với pha 2). Trong ma trận `ma`, các phần tử lớn hơn hoặc bằng `-tol` (ngưỡng chấp nhận) sẽ được mask (ẩn giá trị).

3. Kiểm tra số lượng phần tử không mask trong ma trận `ma`. Nếu số lượng này bằng 0, tức là không có cột phù hợp để pivot, hàm sẽ trả về kết quả False và cột được pivot được đặt là nan.

4. Nếu cờ `bland` được đặt thành True, sử dụng quy tắc Bland để lựa chọn cột pivot. Quy tắc này chọn cột đầu tiên có hệ số âm trong hàng mục tiêu, bất kể giá trị của nó. Hàm sẽ trả về kết quả True và chỉ mục của cột pivot.

5. Nếu không sử dụng quy tắc Bland, hàm tìm chỉ mục của phần tử nhỏ nhất trong ma trận `ma` (chỉ tính các phần tử không mask). Hàm sẽ trả về kết quả True và chỉ mục của cột pivot.

e) Hàm `_pivot_row` :

Hàm `_pivot_row` được sử dụng để xác định biến ra cơ sở.

Quá trình xử lý chi tiết của hàm `_pivot_row`:

1. T tương tự bên trên.

2. Xác định biến `k` dựa trên thuật toán đang ở phase 1 hay phase 2. Nếu `phase` là 1, `k` được đặt thành 2. Nếu `phase` là 2, `k` được đặt thành 1. Mục đích là chỉ xét các hàng không phải hàm mục tiêu.

3. Tạo một ma trận giả mặc định (masked array) `ma` từ cột pivot của ma trận `T` (trừ `k` hàng cuối cùng). Trong ma trận `ma`, các phần tử nhỏ hơn hoặc bằng `tol` (ngưỡng chấp nhận) sẽ được mask (ẩn giá trị).

4. Kiểm tra số lượng phần tử không mask trong ma trận `ma`. Nếu số lượng này bằng 0, tức là không có hàng phù hợp để pivot, hàm sẽ trả về kết quả False và hàng pivot được đặt là nan.

5. Tạo một ma trận giả mặc định `mb` từ cột cuối cùng của ma trận `T` (trừ `k` hàng cuối cùng) tương ứng với hệ số tự do của các ràng buộc. Trong ma trận `mb`, các phần tử tương ứng với các phần tử mask trong `ma` sẽ được mask.

6. Tính vector `q` bằng cách chia ma trận `mb` cho ma trận `ma`. Các phần tử trong `q` tương ứng với các phần tử mask trong `ma` sẽ có giá trị nan.

7. Xác định các chỉ mục của các hàng có giá trị nhỏ nhất trong vector `q` (chỉ tính các phần tử không mask). Đối với quy tắc Bland, chọn chỉ mục có giá trị nhỏ nhất trong danh sách này và tương ứng với biến có chỉ mục nhỏ nhất trong `basis`. Đối với quy tắc không Bland, chỉ trả về chỉ mục đầu tiên trong danh sách.

Nói tóm lại mục tiêu của hàm là tìm hàng có b/a nhỏ nhất, nếu có nhiều hàng có b/a bằng nhau mà cờ `Bland` bằng True thì tìm hàng tương ứng có biến chỉ mục nhỏ nhất trong `basis`

f) Hàm `_apply_pivot` :

Hàm `_apply_pivot` được sử dụng để thực hiện phép xoay dựa vào pivrow và pivcol đã tìm được.
Quá trình xử lý chi tiết của hàm `_apply_pivot`:

1. Nhận đầu vào là ma trận `T`, danh sách `basis` chứa chỉ mục của các biến cơ sở hiện tại, chỉ mục `pivrow` của hàng pivot, và chỉ mục `pivcol` của cột pivot.

2. Cập nhật giá trị trong `basis` tại vị trí `pivrow` thành `pivcol`, đồng nghĩa với việc đổi biến cơ sở từ `basis[pivrow]` sang `pivcol`.

3. Lưu giá trị pivot ban đầu trong biến `pivval` bằng cách truy cập `T[pivrow, pivcol]`.

4. Chia hàng pivot `pivrow` của ma trận `T` cho giá trị pivot `pivval`. Điều này đảm bảo giá trị pivot là 1.

5. Duyệt qua tất cả các hàng trong ma trận `T` (ngoại trừ hàng pivot `pivrow`).

6. Cập nhật hàng hiện tại `T[irow]` bằng cách trừ đi hàng pivot `T[pivrow]` nhân với giá trị tại cột pivot `T[irow, pivcol]`.

g) **Hàm `_solve_simplex` :**

Hàm `_solve_simplex` thực hiện thuật toán đơn hình 2 pha để giải bài toán quy hoạch tuyến tính dựa trên T và danh sách `basis` các biến cơ sở.

Quá trình xử lý chi tiết của hàm `_solve_simplex`:

1. Khởi tạo biến `nit` để đếm số lần lặp và gán giá trị ban đầu cho các biến `status`, `message`, `complete`.

2. Kiểm tra xem `phase` có phải là 1 hoặc 2 không. Nếu không, báo lỗi ValueError.

3. Nếu `phase` là 2, kiểm tra xem có biến nhân tạo nào còn trong `basis` không. Nếu có, kiểm tra xem có hệ số khác không nào từ hàng này và một cột tương ứng với một biến không phải nhân tạo không. Nếu tìm thấy, thực hiện phép pivot tại vị trí đó. Nếu không tìm thấy, bắt đầu `phase` 2 bằng cách tiếp tục vòng lặp.

4. Bắt đầu vòng lặp cho đến khi điều kiện `complete` trở thành `True`.

5. Tìm cột pivot bằng cách gọi hàm `_pivot_col`. Nếu không tìm thấy cột pivot, đánh dấu `status` là 0 và kết thúc vòng lặp. Trường hợp này có nghĩa là tất cả các hệ số của biến không cơ sở trong hàm mục tiêu đều dương, đối với pha 1 thì đã sẵn sàng sang pha 2, còn đối với pha 2 thì đã có thể tìm nghiệm tối ưu.

6. Nếu tìm thấy cột pivot, tiếp tục tìm hàng pivot bằng cách gọi hàm `_pivot_row`. Nếu không tìm thấy hàng pivot, đánh dấu `status` là 3 và kết thúc vòng lặp. Trường hợp này là có vào nhưng không có ra, tương ứng với bài toán vô nghiệm ở pha 1, và bài toán không giới nội (giá trị tối ưu = -00) ở pha 2.

7. Nếu tìm thấy cả cột pivot và hàng pivot, kiểm tra số lần lặp `nit` có vượt quá giới hạn `maxiter` không. Nếu vượt quá, đánh dấu `status` là 1 và kết thúc vòng lặp. Trường hợp này là bài toán bị vòng lặp, sau một số bước xoay thì bải toàn trở lại ban đầu.

8. Nếu chưa vượt quá giới hạn số lần lặp, thực hiện phép pivot bằng cách gọi hàm `_apply_pivot`, tăng `nit` lên 1 và tiếp tục vòng lặp.

9. Kết thúc vòng lặp, trả về giá trị `nit` (số lần lặp) và `status` (trạng thái kết quả của thuật toán).

Ý nghĩa của các status được trình bày trong hàm _linprog_simplex

h) Hàm `_linprog_simplex` :

Giải thích ý nghĩa của từng phần trong hàm:

Ý nghĩa của các biến status :
0 : Thuật toán thành công, có thể ở pha 1 hoặc pha 2. Nếu ở pha 1 nghĩa là pha 1 thành công, có thể bắt đầu pha 2, nếu ở pha 2 thì có nghĩa là bài toán có nghiệm.
1 : Thuật toán vượt qua số lần lặp cho phép, có nghĩa là có thể bài toán bị vòng lặp, sử dụng phép xoay Bland thay vì xoay Dantzig
2 : Bài toán vô nghiệm
3 : Bài toán không giới nội, giá trị tối ưu = -00

Quá trình xử lý chi tiết của hàm `_linprog_simplex`:

1. `status = 0`: Biến `status` được khởi tạo ban đầu với giá trị 0, đại diện cho trạng thái tối ưu hóa thành công.
2. `messages`: Một từ điển chứa thông báo tương ứng với các trạng thái khác nhau của thuật toán tối ưu hóa.
3. `n, m = A.shape`: Gán giá trị số lượng ràng buộc (constraints) `n` và số lượng biến (variables) `m` dựa trên hình dạng của ma trận ràng buộc `A`.
4. `is_negative_constraint`: Một mảng boolean cho biết ràng buộc nào có giá trị b âm (negative constraint).
5. `A[is_negative_constraint] *= -1` và `b[is_negative_constraint] *= -1`: Đảo ngược dấu của các ràng buộc có giá trị b âm, để chuyển chúng thành ràng buộc có giá trị b không âm. Việc này đảm bảo rằng các hệ b luôn không âm mới có thể thực hiện được các thuật toán bên dưới.
6. `av = np.arange(n) + m` và `basis = av.copy()`: Tạo mảng `av` chứa các biến nhân tạo (artificial variables) và sao chép nó vào mảng `basis`

7. `row_constraints`, `row_objective` và `row_pseudo_objective`: Tạo các hàng trong bảng Simplex dựa trên ràng buộc, hàm mục tiêu (z) và hàm mục tiêu giả (w).
8. `T`: Tạo bảng Simplex bằng cách ghép các hàng lại với nhau.
9. `nit1, status`: Thực hiện pha 1 của phương pháp Simplex bằng cách gọi hàm `_solve_simplex` và lưu kết quả vào `nit1` và `status`.
10. `if abs(T[-1, -1]) < tol`: Kiểm tra xem giá trị của hàm mục tiêu giả (w) có gần bằng 0 hay không.
11. Nếu giá trị hàm mục tiêu giả (w) gần bằng 0 (< tol), có nghĩa là pha 1 thành công.
- Xóa hàng hàm mục tiêu giả (w) khỏi bảng Simplex.
- Xóa các cột biến nhân tạo khỏi bảng Simplex.
12. Nếu giá trị pseudo-objective không gần bằng 0, tức là bài toán vô nghiệm, thiết lập trạng thái `status` là 2 và cập nhật thông báo tương ứng.
13. Nếu trạng thái `status` là 0, tức là pha 1 đã thành công :
- Thực hiện pha 2 của phương pháp Simplex bằng cách gọi hàm `_solve_simplex`.
- Lưu kết quả vào `nit2` và cập nhật trạng thái `status`.
14. Tạo một mảng `solution` có kích thước `n + m` và lưu giá trị của các biến cơ bản vào mảng này.
15. Trả về giá trị biến `x`, trạng thái `status` và thông báo tương ứng từ từ điển `messages` dựa trên trạng thái `status`.

i) Hàm `_postsolve`:

Hàm `_postsolve` được sử dụng để chuyển đổi nghiệm từ bài toán quy hoạch tuyến tính đã chuẩn hóa về nghiệm của bài toán gốc ban đầu. Dưới đây là giải thích ý nghĩa của từng phần trong hàm:

1. `c, A_ub, b_ub,_,_,A_eq, b_eq, bounds, prob = self.lp`: bài toán quy hoạch tuyến tính gốc.
2. `n_x = bounds.shape[0]`: số lượng biến của bài toán gốc.
3. `n_unbounded = 0`: Khởi tạo biến `n_unbounded` với giá trị 0 để đếm số lượng biến không bị chặn.
4. Duyệt qua từng biến và giá trị tương ứng trong `bounds`:
- Nếu biến không có giới hạn (unbounded), giảm giá trị của biến đó bằng giá trị của biến nhân tạo tương ứng.
- Nếu biến có giới hạn, điều chỉnh giá trị của biến để đảm bảo nằm trong giới hạn.
5. Giữ lại các giá trị biến quyết định và loại bỏ các giá trị biến nhân tạo khỏi vector `x`.
6. Tính giá trị hàm mục tiêu (objective function value) bằng cách tính tích vô hướng của vector `x` và vector hệ số `c`.
7. Nếu `prob` bằng 0 tức tìm max, đảo ngược giá trị của hàm mục tiêu.

8. Trả về vector `x` (giải pháp của bài toán gốc), giá trị hàm mục tiêu (`fun`)

IV. Tài liệu tham khảo

1. **Dantzig, George B.** *Linear Programming and Extensions.* 1963.

2. **GS. TSKH. Phan Quốc Khánh, TS. Trần Huệ Nương.** *Quy hoạch tuyến tính, Giáo trình hoàn chỉnh : Lý thuyết cơ bản, Phương pháp đơn hình, Bài toàn mạng, Thuật toán điểm trong.* 2003.