Γραμμική & Συνδυαστική Βελτιστοποίηση Εργασία #1

Αβούρης Λάμπρος

Α.Μ.:1092732

Γραμμική & Συνδυαστική Βελτιστοποίηση

Διδάσκων:Δασκαλάκη Σοφία

Γραμμική & Συνδυαστική Βελτιστοποίηση Εργασία #1

Ασκηση 1.

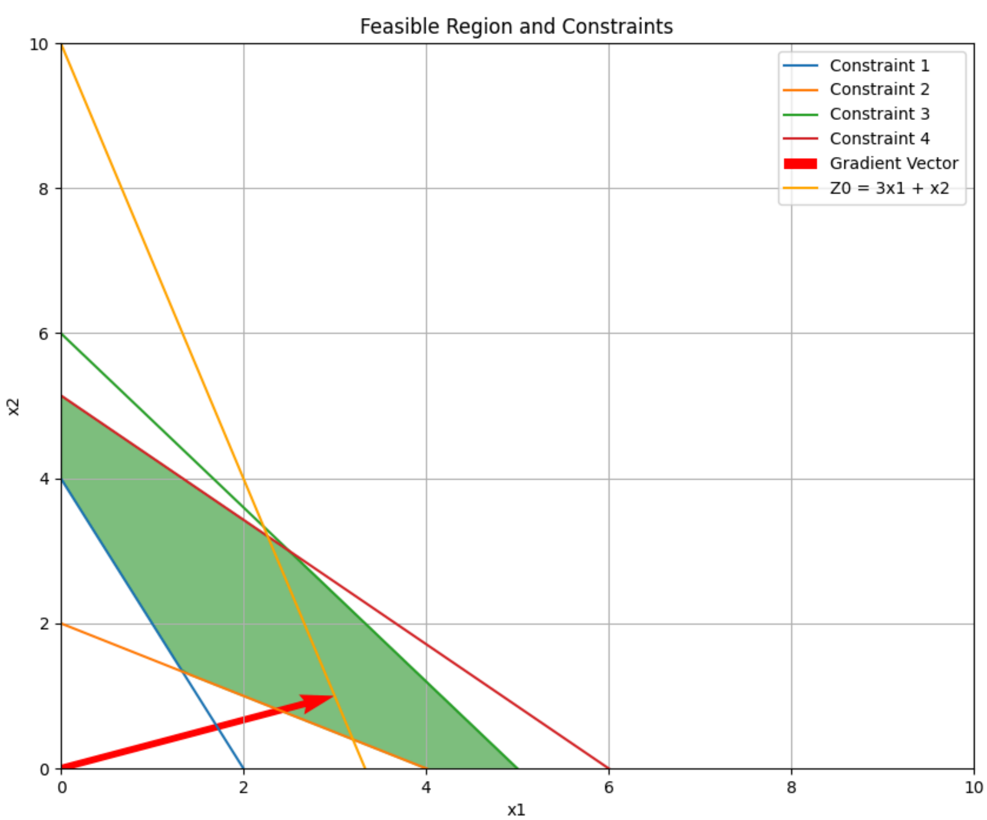
Στην άσκηση αυτή, πρώτων σχεδιάζουμε τις ευθείες τον περιορισμών και μετέπειτα να γραμμοσκιάσουμε την εφικτή περιοχή.

Κατόπιν, δημιουργούμε μια arbitrary Z0 ευθεία, η οποία προφανώς ορίζεται από την αντικειμενική συνάρτηση για δεδομένη τιμη x1 x2.

Μετέπειτα, υπολογίζουμε και αναπαριστούμε το gradient της Z0 ως διάνυσμα που δειχνει προς τα πού μεγιστοποιείτε.

Τέλος, “τραβαμε” νοητά την Z0 , κατα την φορά του διανύσματος. Το τελευταίο vertex εκ του οποίου περνά η ευθεία, θα είναι και η λύση μας.

Προφανώς προκύπτει το [x1,x2] = [5,0] βελτιστη λύση



Γραμμική & Συνδυαστική Βελτιστοποίηση Εργασία #1

Ασκ 1 β)

Για να ισχύει ότι η κορυφή σχηματιζώμενη εκ του Π1 Π2 είναι η βέλτιστη για μια συνάρτηση Ζ θα πρέπει να ίσχύει ότι το gradient της Ζ βρισκετει ανάμεσα στα gradients των δύο περιορισμών, έτσι ώστε οταν “Τραβάμε” την συνλαρτηση Ζ κατα μήκως του gradient να είναι σίγουρο οτι το vertex <Π1, Π2> είναι το τελευταίο το οποίο η συνάρτηση συναντά.

Ανασχηματίζω τους περιορισμούς μου:

$x2 = -2x1 + 4$ (Π1)

$x2 = -0.5x1 + 2$ (Π2)

$x2 = Z - 1/c2 * x1$

Εύκολα υπολογίζω τα gradients

Π1 gradient = -2

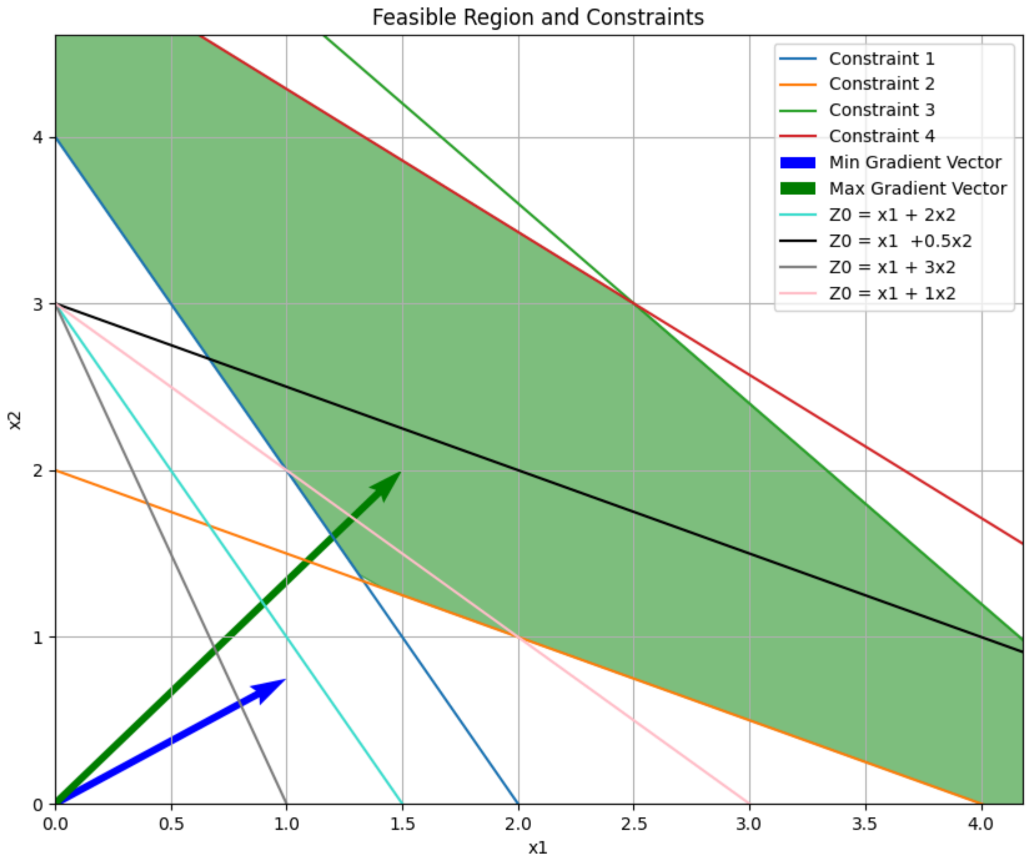
Π2 gradient -0.5

Z gradient = -1/C2

Αρα απο τα παραπάνω έχουμε : $0.5 < C2 < 2$

Μπορώ να επαληθεύσουμε τα παραπάνω μεσω της επακόλουθης γραφικής, όπου έχουν σχεδιαστεί διαφορα Ζ για 4 διαφορετικά C, στο οριο(οπου έχω δυο βέλτιστες λύσεις) εντος(η ζητουμενη κορυφή είναι μοναδική βελτιστη λύση) και εκτός(η κορυφή δεν είναι βέλτιστη λύση) του διαστίματος [0.5,2] για κοινο Z=3

Σαφώς καθώς η Z ελαχιστοποιείται κινείται αντίθετα του gradient



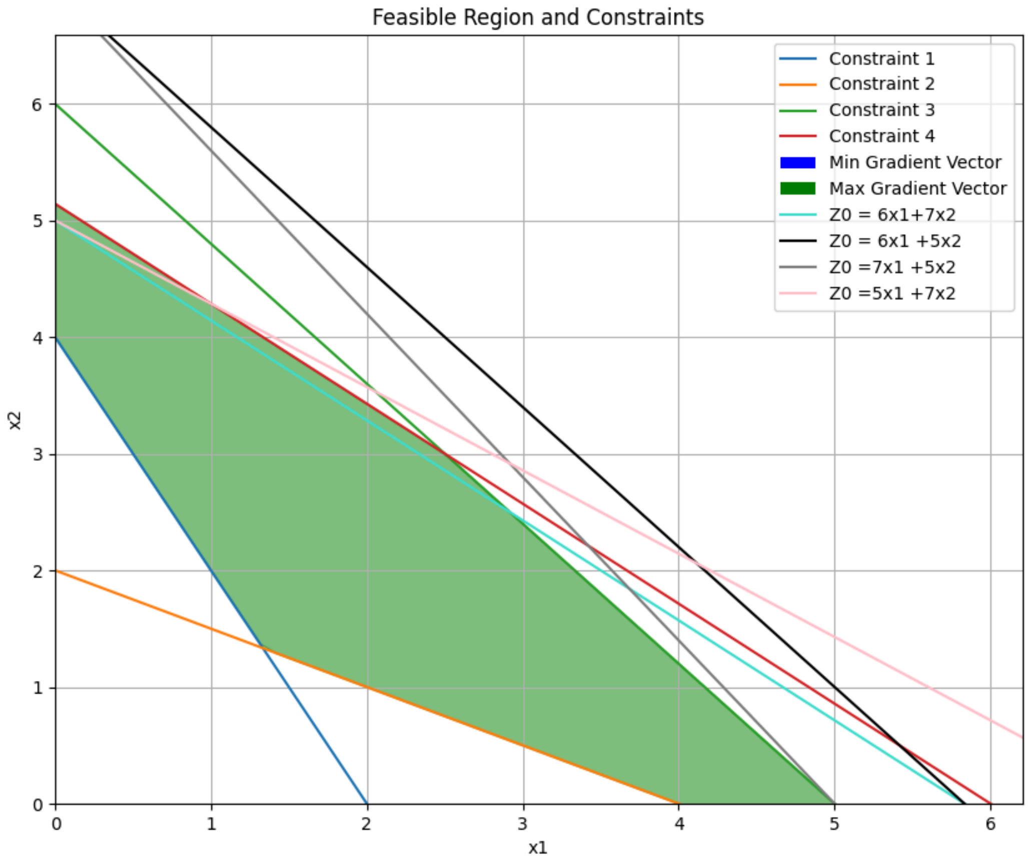
Ασκ 1 γ)

Ομοίως λειτουργώ και στην Γ όπου έχω ως αντίστοιχους περιορισμούς gradient:

$-6/5 < -C1/C2 < -6/7 \Rightarrow 6/5 > C1/C2 > 6/7$(Βρίσκονται εύκολα με την ίδια διαδικασία με παραπάνω)

Άρα για C1, c2 πρεπει να ισχύει $6/5 > C1/C2 > 6/7$

Παρόμοια με παραπάνω έχουμε κάνει plot διάφορα Z για να επαληθεύσουμε τα παραπάνω.



Ο κώδικας για τη παραπάνω ασκηση είναι ο εξής:

```
import numpy as np
import matplotlib.pyplot as plt
f_objective = np.array([3, 1])

A = np.array([[6, 3], [4, 8], [6, 5], [6, 7]])
b = np.array([12, 16, 30, 36])
x1_bounds = (0, None)  # x1 >= 0
x2_bounds = (0, None)  # x2 >= 0
def plot_constraints():
    x1 = np.linspace(0, 10, 100)
    plt.figure(figsize=(10, 8))
```

```
    x2_values = []
    for i in range(A.shape[0]):
        if A[i, 1] != 0:
            x2 = (b[i] - A[i, 0] * x1) / A[i, 1]
            x2_values.append(x2)
            plt.plot(x1, x2, label=f'Constraint {i+1}')
        else:
            plt.axvline(x=b[i]/A[i, 0], label=f'Constraint {i+1}')

    lower_bound = np.maximum(x2_values[0], x2_values[1])
    upper_bound = np.minimum(x2_values[2], x2_values[3])

    # Fill the feasible region
    mask = (lower_bound <= upper_bound)


    plt.fill_between(x1, lower_bound, upper_bound, where=mask,
color='green', alpha=0.5)
    #plot gradient vector
    gradient_vector = np.array([3, 1])
    origin = np.array([0, 0])
    plt.quiver(*origin, *gradient_vector, angles='xy',
scale_units='xy', scale=1, color='red', label='Gradient Vector')

    ## arbitrary  Z0
    Z0 = 10
    x2_line = Z0 - 3 * x1
    plt.plot(x1, x2_line, label='Z0 = 3x1 + x2', color='orange')


    plt.xlim(0, 10)

    plt.ylim(0, 10)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title('Feasible Region and Constraints')
    plt.axhline(0, color='black', lw=0.5)
    plt.axvline(0, color='black', lw=0.5)
```

```python
    plt.grid()
    plt.legend()
    plt.show()

if __name__ == "__main__":
    plot_constraints()
    # We can solve the problem graphicaly by creating an arbitrary
    #  line Z0 = 3x1 + x2
    #and moving it parallel to the vector (3, 1)
    # tHE optimal solution is the last vertex touched by the line.
    #obviosly that is the
    # [x1, x2] = [5,0]
    #C3 and C4  INTERSECTION
    print("the maxixmum is at the intersection of C3 and C4")
    print("x1 = 5, x2 = 0")

    print(f"Z ={5*3+0} ")
```

β)

```python
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
f_objective = np.array([1, 1])
A = np.array([[6, 3], [4, 8], [6, 5], [6, 7]])
b = np.array([12, 16, 30, 36])
x1_bounds = (0, None)  # x1 >= 0
x2_bounds = (0, None)  # x2 >= 0
def plot_constraints():
    x1 = np.linspace(0, 10, 100)
    plt.figure(figsize=(10, 8))

    x2_values = []
    for i in range(A.shape[0]):
        if A[i, 1] != 0:
            x2 = (b[i] - A[i, 0] * x1) / A[i, 1]
            x2_values.append(x2)
            plt.plot(x1, x2, label=f'Constraint {i+1}')
```

```
        else:
            plt.axvline(x=b[i]/A[i, 0], label=f'Constraint {i+1}')

    lower_bound = np.maximum(x2_values[0], x2_values[1])
    upper_bound = np.minimum(x2_values[2], x2_values[3])

    # Fill the feasible region (intersection of all constraints)
    mask = (lower_bound <= upper_bound)


    plt.fill_between(x1, lower_bound, upper_bound,
                    where=mask,
                    color='green', alpha=0.5)
    #plot max and min gradient vector

    P1_gradient =  -np.array([6, 3])
    P2_gradient = -np.array([4, 8])
    origin = np.array([0, 0])
    min_gradient = np.minimum(P1_gradient, P2_gradient)
    max_gradient = np.maximum(P1_gradient, P2_gradient)
    #plot max and min gradient vector
    plt.quiver(*origin, *min_gradient, angles='xy', scale_units='xy',
scale=4, color='blue', label='Min Gradient Vector')
    plt.quiver(*origin, *max_gradient, angles='xy', scale_units='xy',
scale=4, color='green', label='Max Gradient Vector')


    #plt.quiver(*origin, *gradient_vector, angles='xy',
scale_units='xy', scale=1, color='red', label='Gradient Vector')

    Z0 = 3
    x2_line = Z0  -2 * x1
    plt.plot(x1, x2_line, label='Z0 = x1 + 2x2', color='turquoise')
    Z0 = 3
    x2_line2 = Z0 - 0.5 * x1
    plt.plot(x1, x2_line2, label='Z0 = x1  +0.5x2', color='black')
    Z0 = 3
    x2_line3 = Z0 - 3 * x1
    plt.plot(x1, x2_line3, label='Z0 = x1 + 3x2', color='grey')
```

```
    Z0 = 3
    x2_line4 = Z0 - 1 * x1
    plt.plot(x1, x2_line4, label='Z0 = x1 + 1x2', color='pink')
    plt.xlim(0, 10)
    plt.ylim(0, 10)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title('Feasible Region and Constraints')
    plt.axhline(0, color='black', lw=0.5)
    plt.axvline(0, color='black', lw=0.5)
    plt.grid()
    plt.legend()
    plt.show()
```

Στην γ απο τη β αλλάζουν μονο οι συναρτήσεις Z που σχεδιάζουμε αρα δε παραθέτω τον κώδικα.

Ασκ 2. 2

Στην άσκηση 2 καλούμαστε να μοντελοποιήσουμε το παραπάνω πρόβλημα και να το επιλύσουμε γραφικά.

Αρχικά ορίζω τους περιορισμούς ως εξής:

min Z = 0.6x_1 + 0.5x_2

(minimize την ραδιενέργεια στην υγιή περιοχή)

Υπο περιορισμούς ανισότητας:

Ευαίσθητος ιστός: 0.3x_1 + 0.1x_2 <= 2.7

Κεντρο μάζας: 0.6x_1 + 0.4x_2 >= 6

Υπο περιορισμούς ισότητας:

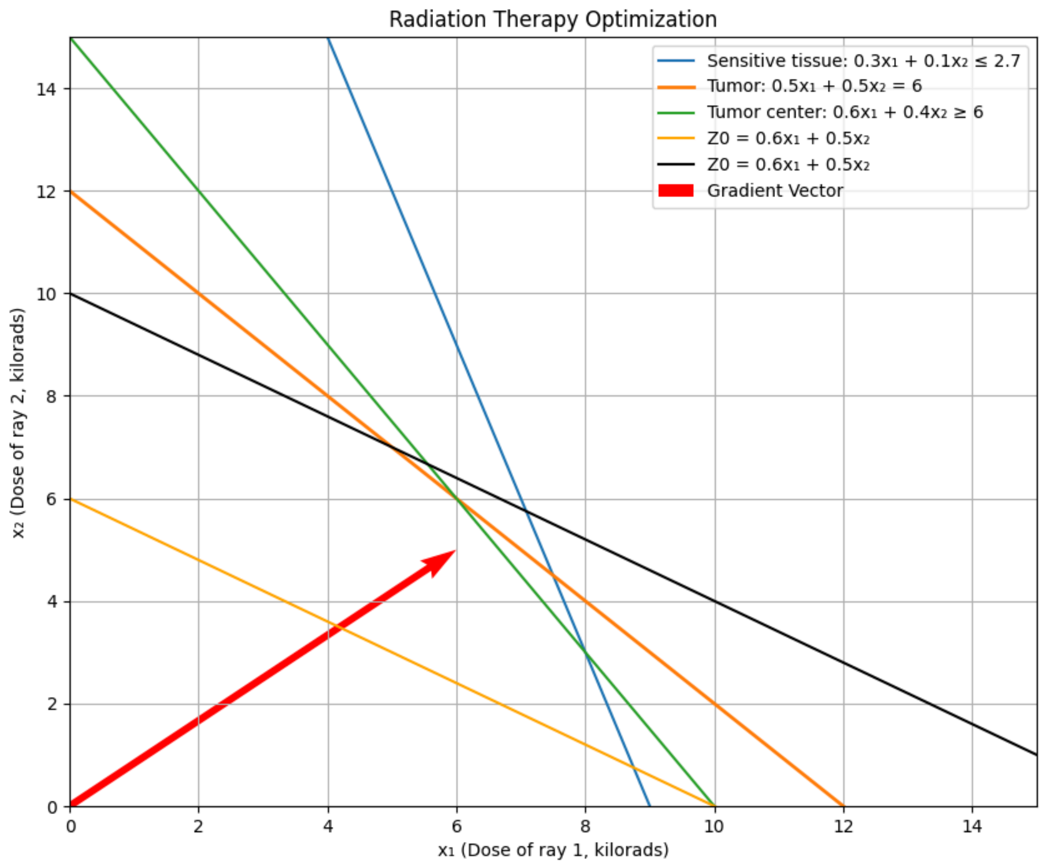
Όγκος: 0.5x_1 + 0.5x_2 = 6

Και περιορισμούς μη μηδενικής τιμής:

x_1 >= 0

X_2 >= 0

Λύνω ακριβώς με την ίδια διαδικασία που έλυσα το πρώτο ερώτημα της πρώτης ασκησης σχεδιάζοντας ένα αυθαίρετο Ζ και μετακινώντας το κατα το gradient του ώστε να βρώ την κατάλληλη κορυφή, μονο που αυτή την φορά η feasible region θα είναι μια ευθεία, λόγω του περιορισμού ισοτητας. Γραφικά προκύπτει το σημείο τομής [x1,x2] = [6,6]



Κώδικας:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import linprog

# Visualize
def plot_solution():
```

```python
    x1 = np.linspace(0, 15, 1000)

    x2_1 = (2.7 - 0.3*x1)/0.1

    x2_2 = 12 - x1

    x2_3 = (6 - 0.6*x1)/0.4

    # Plotting
    plt.figure(figsize=(10, 8))

    # Plot constraints
    plt.plot(x1, x2_1, label='Sensitive tissue: 0.3x₁ + 0.1x₂ ≤ 2.7')
    plt.plot(x1, x2_2, label='Tumor: 0.5x₁ + 0.5x₂ = 6', linewidth=2)
    plt.plot(x1, x2_3, label='Tumor center: 0.6x₁ + 0.4x₂ ≥ 6')

    #plot objective function at Z0 = 6 and Z0 = 10
    Z0 = 6
    x2_line = Z0 - 0.6 * x1
    plt.plot(x1, x2_line, label='Z0 = 0.6x₁ + 0.5x₂', color='orange')
    Z0 = 10
    x2_line2 = Z0 - 0.6 * x1
    plt.plot(x1, x2_line2, label='Z0 = 0.6x₁ + 0.5x₂', color='black')
    # plot the z gradient
    gradient_vector = np.array([0.6, 0.5])
    origin = np.array([0, 0])
    plt.quiver(*origin, *gradient_vector, angles='xy', scale_units='xy',
scale=0.1, color='red', label='Gradient Vector')

    plt.xlim(0, 15)
    plt.ylim(0, 15)
    plt.xlabel('x₁ (Dose of ray 1, kilorads)')
    plt.ylabel('x₂ (Dose of ray 2, kilorads)')
    plt.title('Radiation Therapy Optimization')
    plt.grid(True)
    plt.legend(loc='upper right')
    plt.show()

plot_solution()
```

Γραμμική & Συνδυαστική Βελτιστοποίηση Εργασία #1

Ασκ 4

Μεταβλητές απόφασης θα ορίσω τα κιλά καθε πρώτης ύλης που χρησιμοποιούνται για κάθε ζωοτροφή:

xij όπου i αντιστοιχεί στις πρώτες ύλες, και j στις τροφές αριθμιμένες όπως στο πρόβλημα

i: 1)Καλαμπόκι 2)Ασβεστόλιθος 3)Σόγια 4)Ιχθυάλευρο

j: τροφή I II III

Αντικειμενική συνάρτηση είναι αυτή του κόστους, η οποία προφανώς ελαχιστοποιείται :

$$\min Z = 0.20(x_{11}+x_{12}+x_{13})+0.12(x_{21}+x_{22}+x_{23})+0.24(x_{31}+x_{32}+x_{33})+0.12(x_{41}+x_{42}+x_{43})$$

Εξισώσεις ισότητας προκύπτουν από τις απαιτήσεις για κάθε ζωοτροφή

$$x_{11} + x_{21} + x_{31} + x_{41} = 12000$$

$$x_{12} + x_{22} + x_{32} + x_{42} = 8000$$

$$x_{13} + x_{23} + x_{33} + x_{43} = 9000$$

Εξισώσεις ανισότητας προκύπτουν ως:

1)Περιορισμοί Διαθεσιμότητας Πρώτων Υλών:

$$x_{11} + x_{12} + x_{13} \leq 9000$$

$$x_{21} + x_{22} + x_{23} \leq 12000$$

$$x_{31} + x_{32} + x_{33} \leq 5000$$

$$x_{41} + x_{42} + x_{43} \leq 6000$$

2)Περιορισμοί Θρεπτικών Συστατικών για κάθε ζωοτροφή

Πολλαπλασιάζω με την συνολική ποσότητα της ζωοτροφής ώστε να έχω την συνολική ποσότητα της ουσίας και οχι την ανα kg

Ζωοτροφή Τύπου I (Αγελάδες)

Βιταμίνες/kg ≥ 6:

$$8x_{11} + 6x_{21} + 10x_{31} + 4x_{41} \geq 72000$$

Πρωτεΐνες/kg ≥ 6:

$$10x_{11} + 5x_{21} + 12x_{31} + 8x_{41} \geq 72000$$

Ασβέστιο/kg ≥ 7:

$$6x_{11} + 10x_{21} + 6x_{31} + 6x_{41} \geq 84000$$

4 ≤Λίπος/kg ≤ 8:

$$8x_{11} + 6x_{21} + 6x_{31} + 9x_{41} \geq 48000$$

$$8x_{11} + 6x_{21} + 6x_{31} + 9x_{41} \leq 96000$$

Ζωοτροφή Τύπου II (Πρόβατα)

Βιταμίνες/kg ≥ 6:

$$8x_{12} + 6x_{22} + 10x_{32} + 4x_{42} \geq 48000$$

Πρωτεΐνες/kg ≥ 6:

$$10x_{12} + 5x_{22} + 12x_{32} + 8x_{42} \geq 48000$$

Ασβέστιο/kg ≥ 6:

$$6x_{12} + 10x_{22} + 6x_{32} + 6x_{42} \geq 48000$$

4 ≤ Λίπος/kg ≤ 6:

$$8x_{12} + 6x_{22} + 6x_{32} + 9x_{42} \geq 32000$$

$$8x_{12} + 6x_{22} + 6x_{32} + 9x_{42} \leq 48000$$

Ζωοτροφή Τύπου ΙΙΙ (Κοτόπουλα)

4 ≤ βιταμίνες/kg ≤ 6:

$$8x_{13} + 6x_{23} + 10x_{33} + 4x_{43} \geq 36000$$

$$8x_{13} + 6x_{23} + 10x_{33} + 4x_{43} \leq 54000$$

Πρωτεΐνες/kg ≥ 6:

$$10x_{13} + 5x_{23} + 12x_{33} + 8x_{43} \geq 54000$$

Ασβέστιο ≥ 6

$$6x_{13} + 10x_{23} + 6x_{33} + 6x_{43} \geq 54000$$

4 ≤ λίπος/kg ≤ 5:

$$8x_{13} + 6x_{23} + 6x_{33} + 9x_{43} \geq 36000$$

$$8x_{13} + 6x_{23} + 6x_{33} + 9x_{43} \leq 45000$$

3) Μη Αρνητικότητα

Όλες οι μεταβλητές απόφασης >= 0

Ασκηση 4:

(α) {(x1, x2) ∈ R 2 | x 2 1 + x 2 2 ≥ 3}

Αν πάρουμε τα σημεία $(2, 0)$ και $(-2, 0)$, το ευθύγραμμο τμήμα μεταξύ τους περνάει από το $(0, 0)$, το οποίο δεν ανήκει στο σύνολο. Άρα, το σύνολο ***δεν είναι κυρτό***

(β) {(x1, x2, x3) ∈ R 3 | x1 + 2x2 ≤ 1, x1 − 2x3 ≤ 2}

Το σύνολο $\{(x_1, x_2, x_3) \in \mathbb{R}^3 \mid x_1 + 2x_2 \leq 1, x_1 - 2x_3 \leq 2\}$ είναι η τομή δύο ημιχώρων.

Οι ημιχώροι είναι κυρτά σύνολα, και η τομή κυρτών συνόλων ***είναι κυρτή***.

(γ) {(x1, x2, x3) ∈ R 3 | x2 ≥ x 2 1 , x1 + 2x2 + x3 ≤ 4}

Το σύνολο $\{(x_1, x_2, x_3) \in \mathbb{R}^3 \mid x_2 \geq x_1^2, x_1 + 2x_2 + x_3 \leq 4\}$ είναι η τομή της περιοχής πάνω από μια παραβολή (που κυρτό συνολο) και ενός ημιχώρου (που είναι κυρτό συνολο).

Επομένως το συνολο ***είναι κυρτό*** .

(δ) {(x1, x2, x3) ∈ R 3 |x3 = |x2| , x1 ≤ 3}

Αν πάρουμε τα σημεία $(0, 1, 1)$ και $(0, -1, 1)$, το μέσο $(0, 0, 1)$ δεν ανήκει στο σύνολο, αφού $1 \neq |0|$. Άρα, το σύνολο ***δεν είναι κυρτό***

Γραμμική & Συνδυαστική Βελτιστοποίηση Εργασία #1

Ασκηση 5

Για την άσκηση 5 πρέπει αρχικά να ορίσουμε κάθε περιορισμό ώς ημιεπίπεδο.

Αυτό θα το πετύχουμε κατασκευάζοντας ένα λεξικό που αντιστοιχεί την εξίσωση του κάθε περιορισμού με πίνακα που αντιπροσωπεύει τους συντελεστές, και ενα float που αντιπροσοπεύει την δεξιά μεριά της εξίσωσης.

Μετά κατασκευάζουμε απαραίτητες βοηθητικές συνρτήσεις οι οποίες είναι :

1) ενας solver για γραμμικές εξισώσεις 3 μεταβλητών.
2) Μια συνάρτηση που εφαρμόζει τους περιορισμούς
3) Σαφώς η αντικειμενική συνάρτηση
4) Συνάρτηση που υπολογίζει αν η κορυφή είναι εκφυλισμένη

Έχουμε μετα την εξής αλγοριθμική διαδικασία

Για κάθε συνδυασμό C(set(υπερεπιπέδων), 3):

Επιλύω το σύστημα για να βρω το σημείο x

Εφαρμόζω τους περιορισμούς στο x για να δω αν εφικτό

Βρισκω Z(x)

Υπολογίζω αν το x είναι εκφυλισμένο

Σώζω όλα αυτά τα δεδομένα σε δυο λιστες vertex_info και vertices

Αποθηκεύω μετά τις λιστες μεσω pandas ως dataframe για να μπορώ να τις τυπώσω εύκολα.

Γραμμική & Συνδυαστική Βελτιστοποίηση Εργασία #1

Έχω ως αποτέλεσμα:

Found 32 potential vertices

Of these, 6 are feasible

Feasible vertices (sorted by objective value):

Vertex: (np.float64(3.5), np.float64(6.5), np.float64(8.5))

Formed by: x1 + x2 = 10, x2 + x3 = 15, x1 + x3 = 12

Active constraints: x1 + x2 = 10, x2 + x3 = 15, x1 + x3 = 12

Degenerate: False

Objective value: 94.5 ……. Τα υπόλοιπα vertices

B :

Εδώ πρέπει να τοποθετήσω slack μεταβλητές, και να βρώ για κάθε έναν από τους συνδυασμούς μεταβλητών βασης της αντίστοιχη βασική λύση, και αν η λύση αυτή είναι εφικτή.

Για κάθε συνδιασμό εκτελώ την εξής διαδικασία:

Αρχικά προφανώς ελέγχω ότι η βάση είναι αναστρέψιμη για να μπορώ να προχωρήσω.

Βρίσκω λύση του συστήματος

Ελέγχω αν είναι εφικτή

Ελέγχω αν είναι εκφυλισμένη

Υπολογίζω την αντικειμενική συνάρτηση

Αποθηκεύω με τον ίδιο τρόπο με πρίν τα data σε λίστα και τα επιστρέφω

Μετά εφτιαξα και μια συνάρτηση που τυπώνει τα αποτελέσματα μορφοποιημένα μεσω pandas

Γραμμική & Συνδυαστική Βελτιστοποίηση Εργασία #1

Έχω output :

Original constraints:

$x1 + x2 \geq 10$

$x2 + x3 \geq 15$

$x1 + x3 \geq 12$

$20x1 + 10x2 + 15x3 \leq 300$

With slack variables:

$x1 + x2 - s1 = 10$

$x2 + x3 - s2 = 15$

$x1 + x3 - s3 = 12$

$20x1 + 10x2 + 15x3 + s4 = 300$

$x1, x2, x3, s1, s2, s3, s4 \geq 0$

Found 32 basic solutions

Of these, 6 are feasible and 26 are infeasible

There are 0 degenerate basic solutions

All basic solutions:

Solution 1: FEASIBLE, NON-DEGENERATE

Basis variables: x1, x2, x3, s1

x1 = 6.000000, x2 = 9.000000, x3 = 6.000000

s1 = 5.000000, s2 = 0.000000, s3 = 0.000000, s4 = 0.000000

Objective value: 117.000000 ……. Κλπ

Στο Γ απλά επιλέγω όλες τις βασικές εφικτές λύσεις και τις αντιστοιχώ σε ένα σημείο, ελέγχωντας αν το σημείο αυτό αντιστοιχεί στα [x1,x2,x3] της λύσης.

Έχω αποτέλεσμα:

Vertex (np.float64(3.5), np.float64(6.5), np.float64(8.5)) corresponds to Basic Feasible Solution 1

Basis variables: x1, x2, x3, s4

Objective value: 94.500000

THIS IS THE OPTIMAL SOLUTION….. Κλπ για τις υπόλοιπες

Προφανώς από πάνω καταλαβαίνουμε ότι η βελτιστη κορυφή είναι [3.5,6.5,8.5] και βελτιστη λύση 94.5 ενώ υπάρχουν 6 εφικτές κορυφές και καμία εκφυλισμένη.

```
Κώδικας:
import numpy as np
from itertools import combinations
import pandas as pd
from scipy.optimize import linprog


# Define the constraints as hyperplanes
hyperplanes = [
    {'name': 'x1 + x2 = 10', 'coeffs': [1, 1, 0], 'rhs': 10},
    {'name': 'x2 + x3 = 15', 'coeffs': [0, 1, 1], 'rhs': 15},
    {'name': 'x1 + x3 = 12', 'coeffs': [1, 0, 1], 'rhs': 12},
    {'name': '20x1 + 10x2 + 15x3 = 300', 'coeffs': [20, 10, 15],
'rhs': 300},
    {'name': 'x1 = 0', 'coeffs': [1, 0, 0], 'rhs': 0},
    {'name': 'x2 = 0', 'coeffs': [0, 1, 0], 'rhs': 0},
    {'name': 'x3 = 0', 'coeffs': [0, 0, 1], 'rhs': 0}
]

def solve_system(eq1, eq2, eq3):
```

```
    """Solve  system of 3 linear equations with 3 variables 3x3 
system"""
    A = np.array([eq1['coeffs'], eq2['coeffs'], eq3['coeffs']])
    b = np.array([eq1['rhs'], eq2['rhs'], eq3['rhs']])

    try:
        x = np.linalg.solve(A, b)
        return x
    except np.linalg.LinAlgError:
        return None

def is_feasible(point):
    """Check if a point satisfies all constraints"""
    x1, x2, x3 = point

    #Check n-negativity
    if x1 < -1e-10 or x2 < -1e-10 or x3 < -1e-10:
        return False

    #Check other constraints with a small tolerance
    if x1 + x2 < 10 - 1e-10:
        return False
    if x2 + x3 < 15 - 1e-10:
        return False
    if x1 + x3 < 12 - 1e-10:
        return False
    if 20*x1 + 10*x2 + 15*x3 > 300 + 1e-10:
        return False

    return True

def objective_value(point):
    """Calculate the objective function value"""
    x1, x2, x3 = point
    return 8*x1 + 5*x2 + 4*x3

def check_degeneracy(vertex, hyperplanes_list):
    """Check if a vertex is degenerate """
    active_planes = []
```

```
    for hp in hyperplanes_list:
        x1, x2, x3 = vertex


        a, b, c = hp['coeffs']

        rhs = hp['rhs']
        if abs(a*x1 + b*x2 + c*x3 - rhs) < 1e-10:
            active_planes.append(hp['name'])

    return len(active_planes) > 3, active_planes

# Find all  vertices by solving systems of 3x3 equations
vertices = []
vertex_info = []
for eqs in combinations(hyperplanes, 3):
    point = solve_system(eqs[0], eqs[1], eqs[2])

    if point is not None:
        is_feas = is_feasible(point)
        is_deg, active_planes = check_degeneracy(point, hyperplanes)

        obj_val = objective_value(point) if is_feas else None

        vertex_info.append({
            'point': tuple(np.round(point, 10)),
            'equations': [eq['name'] for eq in eqs],
            'feasible': is_feas,
            'degenerate': is_deg,
            'active_planes': active_planes,
            'objective_value': obj_val
        })

        if is_feas:
            vertices.append(point)

# Sort and display the results
vertex_df = pd.DataFrame(vertex_info)
```

```
feasible_df = vertex_df[vertex_df['feasible']].copy()
feasible_df = feasible_df.sort_values(by='objective_value')

# Part B:
def find_basic_solutions():
    """Find all basic solutions after adding slack variables"""
    # A matrix
    A = np.array([
        [1, 1, 0, -1, 0, 0, 0],
        [0, 1, 1, 0, -1, 0, 0],
        [1, 0, 1, 0, 0, -1, 0],
        [20, 10, 15, 0, 0, 0, 1]
    ])

    # B matrix
    b = np.array([10, 15, 12, 300])

    # Variables
    var_names = ['x1', 'x2', 'x3', 's1', 's2', 's3', 's4']

    basic_solutions = []

    # Try all combinations of 4 variables to be in the basis
    for basis_indices in combinations(range(7), 4):
        # Extract variables columns φορ hte basis
        B = A[:, basis_indices]

        # Try to solve the system
        try:
            # Check if invertible
            if abs(np.linalg.det(B)) > 1e-10:
                #Solve
                basis_values = np.linalg.solve(B, b)

                # Create  solution vector
                solution = np.zeros(7)
                for i, idx in enumerate(basis_indices):
                    solution[idx] = basis_values[i]
```

```
                #Check if feasible
                is_feasible = np.all(solution >= -1e-10)

                #Check if  degen
                zero_basics = sum(1 for i, idx in
enumerate(basis_indices) if abs(basis_values[i]) < 1e-10)
                is_degenerate = zero_basics > 0

                # Calculate Z for feasible solutions
                obj_val = None
                if is_feasible:
                    obj_val = 8*solution[0] + 5*solution[1] +
4*solution[2]

                # Store the solution info
                basic_solutions.append({
                    'basis_variables': [var_names[idx] for idx in
basis_indices],
                    'solution': solution,
                    'feasible': is_feasible,
                    'degenerate': is_degenerate,
                    'objective_value': obj_val,
                    'x1': solution[0],
                    'x2': solution[1],
                    'x3': solution[2],
                    's1': solution[3],
                    's2': solution[4],
                    's3': solution[5],
                    's4': solution[6]
                })
        except np.linalg.LinAlgError:
            # no solution
            continue

    return basic_solutions

def add_slack_variables():
    """Convert the problem to standard form with slack variables """
    print("\nPart B: ")
```

```
    print("Original constraints:")
    print("x1 + x2 >= 10")
    print("x2 + x3 >= 15")
    print("x1 + x3 >= 12")
    print("20x1 + 10x2 + 15x3 ≤ 300")

    print("\nWith slack variables:")
    print("x1 + x2 - s1 = 10")
    print("x2 + x3 - s2 = 15")
    print("x1 + x3 - s3 = 12")
    print("20x1 + 10x2 + 15x3 + s4 = 300")
    print("x1, x2, x3, s1, s2, s3, s4 >= 0")

    # Find all basic solutions
    basic_solutions = find_basic_solutions()

    # Create a DataFrame for better display
    df = pd.DataFrame(basic_solutions)

    # Count solutions
    feasible_count = df['feasible'].sum()
    infeasible_count = len(df) - feasible_count

    # Count degenerate solutions
    degenerate_count = df['degenerate'].sum()

    print(f"\nFound {len(df)} basic solutions")
    print(f"Of these, {feasible_count} are feasible and
{infeasible_count} are infeasible")
    print(f"There are {degenerate_count} degenerate basic solutions")

    # Display all basic solutions
    print("\nAll basic solutions:")
    for i, sol in enumerate(basic_solutions):
        basis_str = ', '.join(sol['basis_variables'])
        status = "FEASIBLE" if sol['feasible'] else "INFEASIBLE"
        degeneracy = "DEGENERATE" if sol['degenerate'] else
"NON-DEGENERATE"
```

```
        print(f"\nSolution {i+1}: {status}, {degeneracy}")
        print(f"Basis variables: {basis_str}")
        print(f"x1 = {sol['x1']:.6f}, x2 = {sol['x2']:.6f}, x3 =
{sol['x3']:.6f}")
        print(f"s1 = {sol['s1']:.6f}, s2 = {sol['s2']:.6f}, s3 =
{sol['s3']:.6f}, s4 = {sol['s4']:.6f}")

        if sol['feasible']:
            print(f"Objective value: {sol['objective_value']:.6f}")

    return basic_solutions

# Part C: Find optimal solution and establish correspondence
def compare_solutions(vertices_df, basic_solutions):
    """Compare vertices from Part A with basic feasible solutions
from Part B"""
    print("\nPart C: Correspondence between vertices and basic
feasible solutions")

    # Filter for feasible basic solutions
    feasible_basic = [sol for sol in basic_solutions if
sol['feasible']]

    #Sort by Z
    feasible_basic.sort(key=lambda x: x['objective_value'] if
x['objective_value'] is not None else float('inf'))

    #Find the optimal solution
    optimal = feasible_basic[0]
    print(f"\nOptimal solution:")
    print(f"x1 = {optimal['x1']:.6f}, x2 = {optimal['x2']:.6f}, x3 =
{optimal['x3']:.6f}")
    print(f"s1 = {optimal['s1']:.6f}, s2 = {optimal['s2']:.6f}, s3 =
{optimal['s3']:.6f}, s4 = {optimal['s4']:.6f}")
    print(f"Objective value: {optimal['objective_value']:.6f}")

    #Createvertices το solutions map
    print("\nCorrespondence between vertices and basic feasible
solutions:")
```

```
    for i, vertex_row in
vertices_df[vertices_df['feasible']].iterrows():
        vertex = vertex_row['point']
        #Find matching basic feasible solution
        for j, basic_feasible_solution in enumerate(feasible_basic):
            if (abs(vertex[0] - basic_feasible_solution['x1']) <
1e-10 and
                abs(vertex[1] - basic_feasible_solution['x2']) <
1e-10 and
                abs(vertex[2] - basic_feasible_solution['x3']) <
1e-10):

                print(f"\nVertex {vertex} corresponds to Basic
Feasible Solution {j+1}")
                print(f"Basis variables: {',
'.join(basic_feasible_solution['basis_variables'])}")
                print(f"Objective value:
{basic_feasible_solution['objective_value']:.6f}")

                if j == 0:
                    print("THIS IS THE OPTIMAL SOLUTION")
                break

# Display ----------------------------
print("Part A:")
print(f"Found {len(vertex_info)} potential vertices")
print(f"Of these, {len(vertices)} are feasible")

print("\nFeasible vertices (sorted by objective value):")
for idx, row in feasible_df.iterrows():
    print(f"Vertex: {row['point']}")
    print(f"  Formed by: {', '.join(row['equations'])}")
    print(f"  Active constraints: {', '.join(row['active_planes'])}")
    print(f"  Degenerate: {row['degenerate']}")
    print(f"  Objective value: {row['objective_value']}")
    print()

basic_solutions = add_slack_variables()
compare_solutions(vertex_df, basic_solutions)
```

Ασκ 6.

Για την υλοποίηση simplex αρχικά χρειαζόμαστε τις εξής βοηθητικές συναρτήσεις:

1) Display function για να τυπώνει τα tableaux
2) Function που βρίσκει όλες τις πιθανές εισερχόμενες μεταβλητές βρίσκει δηλαδή αυτές που έχουν συντελεστή <0
3) Function που βρίσκει όλες τις πιθανές εξερχόμενες μεταβλητές μεσω minimum ratio test
4) Function που ελέγχει αν είναι optimal η λύση
5) Function που εκτελεί το pivot, αλλάζοντας την εξερχόμενη με εισερχόμενη και κάνει update τις τιμές

Μετά απλά εφαρμόζω τον αλγόριθμο simplex ως εξής:

Ωσω η λύση δεν είναι optimal:

Βρες πιθανές εισερχόμενες

Βρες πιθανές εξερχόμενες

Αν δεν υπάρχουν εξερχόμενες το πρόβλημα είναι unbounded, break

Διάλεξε εισερχόμενη και εξερχόμενη βάση κριτηρίων

Εκτέλεσε pivot

Υπολόγισε την νέα λύση.

Τύπωσε ότι έκανες.

Παίρνω την εξής λύση:

Initial Tableau:

	Basic	Z	x_1	x_2	x_3	x_4	s_1	s_2	s_3	RHS
0	Z	1.0	-2.0	-1.0	-6.0	4.0	0.0	0.0	0.0	0.0
1	s_1	0.0	1.0	2.0	4.0	-1.0	1.0	0.0	0.0	6.0
2	s_2	0.0	2.0	3.0	-1.0	1.0	0.0	1.0	0.0	12.0
3	s_3	0.0	1.0	0.0	1.0	1.0	0.0	0.0	1.0	2.0

……. iterations

Iteration 2:

Potential entering variables: ['x_1']

Selected entering variable: x_1

Potential leaving variables: ['s_2']

Selected leaving variable: s_2

New tableau:

	Basic	Z	x_1	x_2	x_3	x_4	s_1	s_2	s_3	RHS
0	x_3	1.0	0.0	1.666667	0.0	3.333333	1.333333	0.0	0.666667	9.333333
1	s_1	0.0	0.0	0.666667	1.0	-0.666667	0.333333	0.0	-0.333333	1.333333
2	x_1	0.0	0.0	5.000000	0.0	-3.000000	1.000000	1.0	-3.000000	12.000000
3	s_3	0.0	1.0	-0.666667	0.0	1.666667	-0.333333	0.0	1.333333	0.666667

Optimal solution found :) ! Solution:

$x_1 = 12.0$

$x_2 = 0$

$x_3 = 0$

$x_4 = 0$

Objective value: Z = 9.333333333333334

Στο β ο αλγόριθμος παραμένει ο ίδιος, όμως εδώ αντι να επιλέγουμε μια εισερχόμενη και μια εξερχόμενη μεταβλητή, επιλέγουμε όλες τις πιθανές μεταβολές κάθε φορά.

Το επιτυγχάνουμε μέσω nested for loop εξερευνώντας πρώτα εισερχόμενες και μετα εξερχόμενες

Θα εξερευνήσουμε όλα αυτά τα pivots με breadth first αναζήτηση, και κάθε pivot, θα το αντιστοιχίσουμε σε έναν edge του adjacency graph.

Οι λύσεις που προκύπτει ότι συνδέονται απο το pivot αυτό θα αντιστοιχίζονται σε ένα node του adjacency graph το οποίο αν δεν υπάρχει ήδη, το δημιουργούμε

Πρέπει προφανώς, να προσέχουμε να μην ξανα επισκεπτόμαστε πολλές φορές το ίδιο pivot.

Αν βρεθεί εδώ μια optimal λύση, δεν σταματάμε, αλλά συνεχίζουμε μέχρι να μην μπορούμε να κάνουμε νέο pivot

Κάνουμε plot το γραφω μέσω της library networkx και τυπώνουμε τα αποτελέσματα της εξερεύνησης:

Simplex Adjacency Graph Summary:

Total nodes (basic feasible solutions): 6

Total edges (pivot operations): 9

Optimal solutions found: 1

Optimal Solutions:

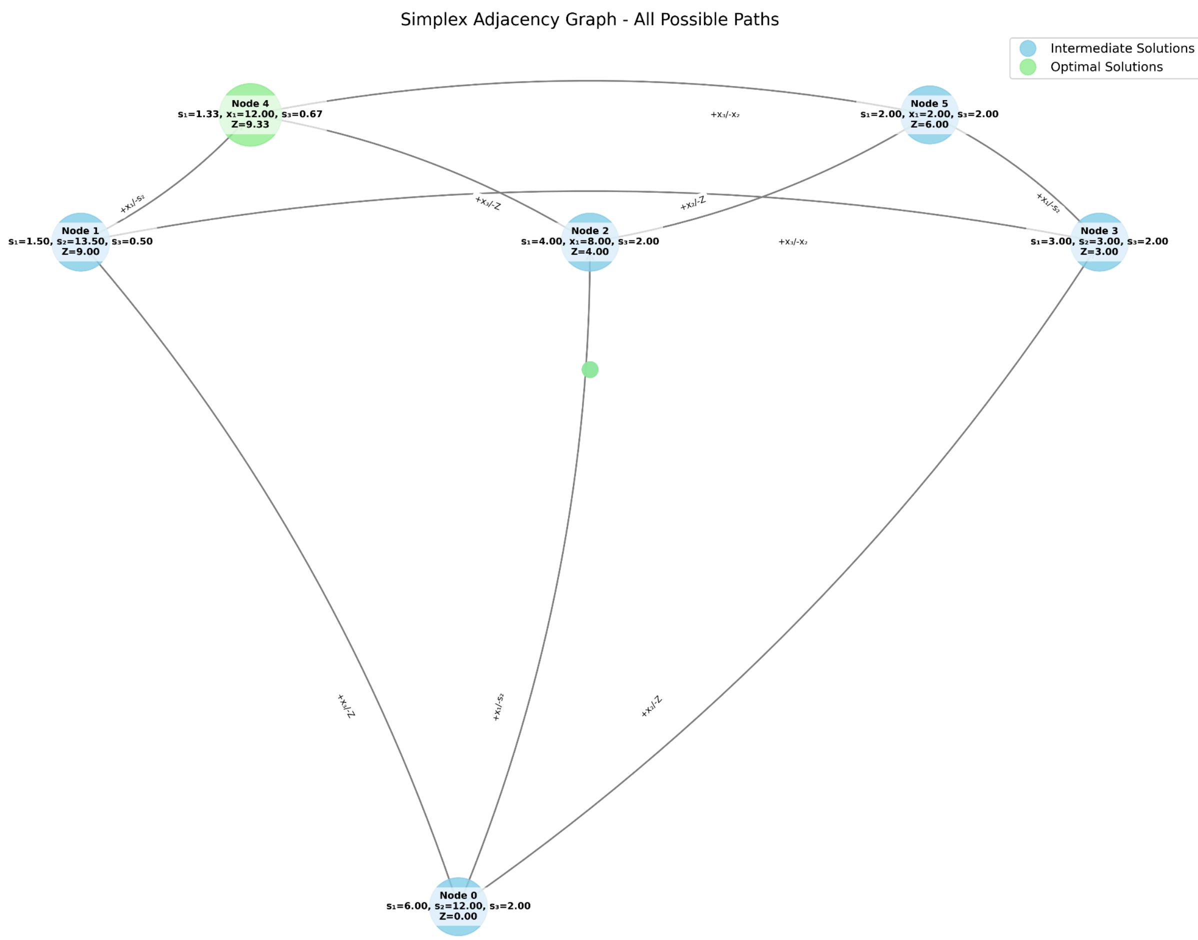
Node 4: Z = 9.333333333333334

Node 4

s_1=1.33, x_1=12.00, s_3=0.67

Z=9.33

Έχω το εξής αποτέλεσμα:



Κώδικας:

```
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque

def create_initial_tableau():
    # Initial tableau with slack variables s1, s2, s3

    return np.array([
```

```
        [1, -2, -1, -6, 4, 0, 0, 0, 0],
        [0, 1, 2, 4, -1, 1, 0, 0, 6],
        [0, 2, 3, -1, 1, 0, 1, 0, 12],
        [0, 1, 0, 1, 1, 0, 0, 1, 2]
    ], dtype=float)
##pretty print our tableaux
def display_tableau(tableau, basic_vars):
    columns = ['Z', 'x₁', 'x₂', 'x₃', 'x₄', 's₁', 's₂', 's₃', 'RHS']
    df = pd.DataFrame(tableau, columns=columns)
    df.insert(0, 'Basic', basic_vars)
    return df

def find_entering_variables(tableau):
    # Find all <0 coefficients in objective row (potential entering vars)
    objective_coeffs = tableau[0, 1:-1]
    negative_indices = np.where(objective_coeffs < 0)[0]
    entering_candidates = [idx + 1 for idx in negative_indices]

    # Sort by absolute value
    return sorted(entering_candidates, key=lambda x: tableau[0, x])

def find_leaving_variables(tableau, entering_col):
    # Calculate ratios for minimum ratio test
    ratios = []
    for i in range(1, len(tableau)):
        if tableau[i, entering_col] <= 0:
            ratios.append((i, float('inf')))
        else:
            ratios.append((i, tableau[i, -1] / tableau[i, entering_col]))

    # Filter out ινφ ratios (unbounded)
    valid_ratios = [r for r in ratios if r[1] != float('inf')]

    if not valid_ratios:
        return []  # No  leaving variables (unbounded)

    min_ratio = min(r[1] for r in valid_ratios)
```

```
    # Find all rows with minimum ratio (potential leaving variables)
    leaving_candidates = [row for row, ratio in ratios if ratio ==
min_ratio]

    return leaving_candidates

def pivot(tableau, basic_vars, entering_col, leaving_row):

    new_tableau = tableau.copy()
    new_basic_vars = basic_vars.copy()

    # Get the pivot element
    pivot_element = tableau[leaving_row, entering_col]

    #Update basic
    col_names = ['Z', 'x₁', 'x₂', 'x₃', 'x₄', 's₁', 's₂', 's₃']
    new_basic_vars[leaving_row-1] = col_names[entering_col]

    # Normalize pivot
    new_tableau[leaving_row] = tableau[leaving_row] / pivot_element

    # Update  other rows
    for i in range(len(tableau)):
        if i != leaving_row:
            factor = tableau[i, entering_col]
            new_tableau[i] = tableau[i] - factor *
new_tableau[leaving_row]

    return new_tableau, new_basic_vars

def is_optimal(tableau):
    # Check if all coefficients in objective row are non-negative
    objective_coeffs = tableau[0, 1:-1]
    return np.all(objective_coeffs >= 0)

def solve_simplex():
    tableau = create_initial_tableau()
    basic_vars = ['Z', 's₁', 's₂', 's₃']
```

```python
    print("Initial Tableau:")
    print(display_tableau(tableau, basic_vars))

    iteration = 0
    symbols_lst = ['Z', 'x₁', 'x₂', 'x₃', 'x₄', 's₁', 's₂', 's₃']
    while not is_optimal(tableau):
        iteration += 1
        print(f"\nIteration {iteration}:")

        # Find potential entering variables
        entering_candidates = find_entering_variables(tableau)
        print(f"Potential entering variables: {[symbols_lst[col] for col in entering_candidates]}")

        # Select the entering variable (most negative )
        entering_col = entering_candidates[0]
        entering_var = symbols_lst[entering_col]
        print(f"Selected entering variable: {entering_var}")

        #Find potential leaving vars
        leaving_candidates = find_leaving_variables(tableau, entering_col)

        if not leaving_candidates:
            print("Problem is unbounded!")
            return

        leaving_vars = [basic_vars[row-1] for row in leaving_candidates]
        print(f"Potential leaving variables: {leaving_vars}")

        #Select the leaving variable ,first candidate no special evaluation
        leaving_row = leaving_candidates[0]
        leaving_var = basic_vars[leaving_row-1]
        print(f"Selected leaving variable: {leaving_var}")

        # Perform pivot
```

```
        tableau, basic_vars = pivot(tableau, basic_vars,
entering_col, leaving_row)
        print("New tableau:")
        print(display_tableau(tableau, basic_vars))

    # Extract solution
    print("\nOptimal solution found :)!")
    solution = {}
    for i, var in enumerate(basic_vars[1:]):  # Skip Z
        solution[var] = tableau[i+1, -1]

    # Set non-basic  to zero
    all_vars = ['x₁', 'x₂', 'x₃', 'x₄', 's₁', 's₂', 's₃']
    for var in all_vars:
        if var not in basic_vars:
            solution[var] = 0

    print("\nSolution:")
    for var in ['x₁', 'x₂', 'x₃', 'x₄']:
        print(f"{var} = {solution.get(var, 0)}")

    print(f"\nObjective value: Z = {tableau[0, -1]}")

def explore_all_paths():
    # Initialize a directed graph
    G = nx.DiGraph()

    # Create initial tableau and basic variables
    initial_tableau = create_initial_tableau()
    initial_basic_vars = ['Z', 's₁', 's₂', 's₃']

    # Use a queue for breadth-first exploration
    # (tableau, basic_vars, iteration)
    queue = deque([(initial_tableau, initial_basic_vars, 0)])
     # Track visited states
    visited = set()
    # Map state keys to node IDs for better visualization

    node_map = {}
```

```
    while queue:
        current_tableau, current_basic_vars, iteration = 
queue.popleft()

        # Create a key for the current state
        values = 
[f"{current_basic_vars[i+1]}={current_tableau[i+1,-1]:.2f}"
                 for i in range(len(current_basic_vars)-1)]
        state_key = tuple(values)

        # Skip if already visited
        if state_key in visited:
            continue

        # Mark as visited
        visited.add(state_key)

        # Generate a node if needed
        if state_key not in node_map:
            node_id = len(node_map)
            node_map[state_key] = node_id

            # Create node label
            basic_vars_str = ", ".join(values)
            z_value = current_tableau[0, -1]
            node_label = f"Node 
{node_id}\n{basic_vars_str}\nZ={z_value:.2f}"

            # Add node to graph
            G.add_node(node_id, label=node_label, 
iteration=iteration,
                       optimal=is_optimal(current_tableau), 
z_value=z_value)

        # Get current node ID
        current_node_id = node_map[state_key]

        # If solution is optimal, mark it and continue
        if is_optimal(current_tableau):
```

```
            G.nodes[current_node_id]['optimal'] = True
            continue

        # Find all possible entering
        entering_candidates =
find_entering_variables(current_tableau)

        # Explore each entering variable option
        for entering_col in entering_candidates:
            entering_var = ['Z', 'x₁', 'x₂', 'x₃', 'x₄', 's₁', 's₂',
's₃'][entering_col]

            # Find all possible leaving variables
            leaving_candidates =
find_leaving_variables(current_tableau, entering_col)

            if not leaving_candidates:
                # Unbounded problem for this path
                continue

            # Explore each leaving variable
            for leaving_row in leaving_candidates:
                leaving_var = current_basic_vars[leaving_row-1]

                # Create new tableau and basic variables after pivot
                new_tableau, new_basic_vars = pivot(
                    current_tableau, current_basic_vars.copy(),
                    entering_col, leaving_row
                )

                # Add to queue
                queue.append((new_tableau, new_basic_vars, iteration
+ 1))

                # Create key for the next state
                new_values =
[f"{new_basic_vars[i+1]}={new_tableau[i+1,-1]:.2f}"
                             for i in range(len(new_basic_vars)-1)]
                new_state_key = tuple(new_values)
```

```
                # Create node for the next state if needed
                if new_state_key not in node_map:
                    new_node_id = len(node_map)
                    node_map[new_state_key] = new_node_id

                    # Create node label
                    new_basic_vars_str = ", ".join(new_values)
                    new_z_value = new_tableau[0, -1]
                    new_node_label = f"Node
{new_node_id}\n{new_basic_vars_str}\nZ={new_z_value:.2f}"

                    # Add node to graph
                    G.add_node(new_node_id, label=new_node_label,
iteration=iteration+1,
                              optimal=is_optimal(new_tableau),
z_value=new_z_value)

                # Get next node ID
                next_node_id = node_map[new_state_key]

                # Add edge between nodes
                G.add_edge(current_node_id, next_node_id,
                          label=f"+{entering_var}/-{leaving_var}")

    # Draw the graph
    plt.figure(figsize=(18, 14))  # Larger figure size


    pos = nx.spring_layout(G, k=0.5, iterations=100, seed=42)

    # Adjust node positions to prevent overlap
    iteration_levels = {}
    for node, data in G.nodes(data=True):
        level = data.get('iteration', 0)
        if level not in iteration_levels:
            iteration_levels[level] = []
        iteration_levels[level].append(node)
```

```
    # Adjust positions based on iteration levels
    for level, nodes in iteration_levels.items():
        if len(nodes) > 1:
            # Spread nodes at same level horizontally
            base_y = 0.2 * level
            spacing = 1.0 / (len(nodes) + 1)
            for i, node in enumerate(nodes):
                x_pos = -0.5 + (i + 1) * spacing
                pos[node] = np.array([x_pos, base_y])

    # Draw nodes with different colors
    optimal_nodes = [n for n, d in G.nodes(data=True) if
d.get('optimal', False)]
    other_nodes = [n for n in G.nodes() if n not in optimal_nodes]

    nx.draw_networkx_nodes(G, pos, nodelist=other_nodes,
                          node_color='skyblue', node_size=3000,
alpha=0.8)
    nx.draw_networkx_nodes(G, pos, nodelist=optimal_nodes,
                          node_color='lightgreen', node_size=3500,
alpha=0.8)

    # Draw edges with arrows
    nx.draw_networkx_edges(G, pos, arrows=True, width=1.5,
                          arrowsize=15, arrowstyle='->',
edge_color='gray',
                          connectionstyle='arc3,rad=0.1')  # Curved
edges to avoid overlap

    # Draw node labels
    node_labels = {n: d['label'] for n, d in G.nodes(data=True)}
    nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=9,
                           font_family='sans-serif',
font_weight='bold',
                           bbox=dict(facecolor='white', alpha=0.7,
edgecolor='none', pad=4))

    # Draw edge labels
    edge_labels = {(u, v): d['label'] for u, v, d in
```

```
G.edges(data=True)}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_size=8,
                                label_pos=0.3)  # Adjust label
position on edge

    # Add  legend
    plt.plot([0], [0], 'o', color='skyblue', label='Intermediate
Solutions', ms=15, alpha=0.8)
    plt.plot([0], [0], 'o', color='lightgreen', label='Optimal
Solutions', ms=15, alpha=0.8)
    plt.legend(loc='best', fontsize=12)

    # Add  title
    plt.title('Simplex Adjacency Graph - All Possible Paths',
fontsize=16)
    plt.axis('off')
    plt.tight_layout()


    plt.subplots_adjust(left=0.05, right=0.95, top=0.95, bottom=0.05)

    # Save and show  graph
    plt.savefig('simplex_adjacency_graph.png', dpi=300,
bbox_inches='tight')
    plt.show()

    # Print summary
    print("\nSimplex Adjacency Graph Summary:")
    print(f"Total nodes (basic feasible solutions):
{len(G.nodes())}")
    print(f"Total edges (pivot operations): {len(G.edges())}")
    print(f"Optimal solutions found: {len(optimal_nodes)}")

    # List all optimal solutions
    print("\nOptimal Solutions:")
    for node in optimal_nodes:
        z_value = G.nodes[node]['z_value']
        print(f"Node {node}: Z = {z_value}")
```

```
        print(G.nodes[node]['label'])
        print("---")

# Main script execution
if __name__ == "__main__":
    # First run standard simplex
    print("Part (a): Step-by-step execution of Simplex algorithm")
    solve_simplex()

    # Then explore all possible paths
    print("\n\nPart (b): Exploring all possible Simplex paths")
    explore_all_paths()
```