HungarianMethod

May 6, 2021

1 Zagadnienie przydziału - metoda węgierska

Grupa: 4a, czwartek 14:30 – 16:00

Data wykonania: 29.04.2021

Skład zespołu:

-Zuzanna Zielińska (Krok 3 - Sprawdzenie liczby zer niezależnych)

-Zofia Lenarczyk (Krok 1 - Redukcja macierzy)

-Maciej Kucharski (Krok 4 - Próba powiększenia zbioru zer niezależnych)

-Łukasz Rams (Krok 2 - Poszukiwanie kompletnego przydziału)

Celem ćwiczenia było zespołowe zaimplementowanie metody węgierskiej - algorytmu rozwiązującego problem przydziału.

1.0.1 Importy potrzebnych bibliotek

[8]:
```
import numpy as np
from typing import Tuple, Optional, Union, Tuple
from collections import Counter
from itertools import product
```

1.1 Krok 1 - Redukcja macierzy

[9]:
```
def reduce_matrix(matrix: np.ndarray) -> Optional[Tuple[np.ndarray, int]]:

    """
    Funkcja realizuje redukcję macierzy kwadratowej.
    W każdym wierszu wyszukiwany jest element najmniejszy, który następnie jest␣
 ↪dodawany do kosztu redukcji oraz
    odejmowany (redukowany) od wszystkich elementów w wierszu.
    Analogiczne działanie dla kolumn macierzy.

    :param matrix: macierz wejściowa
    :return: zredukowana macierz, koszt redukcji
    """

```

```
    # Sprawdzenie czy macierz jest kwadratowa:
    if matrix.shape[0] != matrix.shape[1]:
        print('Niepoprawne wymiary macierzy wejściowej')
        return None

    reduction_cost: int = 0  # Koszt redukcji
    reduced_rows_matrix: np.ndarray = np.zeros(matrix.shape)  # Macierz z␣
↪zredukowanymi wierszami
    reduced_matrix: np.ndarray = np.zeros(matrix.shape)  # Zredukowana macierz

    # Redukcja wierszy
    for i in range(matrix.shape[0]):
        minimum = matrix[i].min()
        reduction_cost += minimum
        row_reduced = [x - minimum for x in matrix[i]]
        reduced_rows_matrix[i] = row_reduced

    print(f'\nMacierz ze zredukowanymi wierszami:\n {reduced_rows_matrix}')
    print(f'Koszt redukcji wierszy: {reduction_cost}')

    # Redukcja kolumn
    for i in range(matrix.shape[0]):
        minimum = reduced_rows_matrix[:, i].min()
        reduction_cost += minimum
        col_reduced = [x - minimum for x in reduced_rows_matrix[:, i]]
        reduced_matrix[:, i] = col_reduced

    print(f'\nZredukowana macierz:\n {reduced_matrix}')
    print(f'Całkowity koszt redukcji macierzy: {reduction_cost}')

    return set_zeros(reduced_matrix, reduction_cost)
```

Komentarz Wynik redukcji jest zależny od kolejności jej wykonywania. Macierz zredukowana oraz koszt redukcji będzie się różnić w zależności czy najpierw wykonana jest redukcja wierszy czy kolumn.

1.2 Krok 2 - Poszukiwanie kompletnego przydziału

[10]:
```
def set_zeros(matrix: np.ndarray, fi: Union[int, float]) -> Tuple[np.ndarray,␣
↪np.ndarray, Union[int, float]]:
    """
  ␣
↪===========================================================================================
    Slajd 2
    Poszukiwanie kompletnego przydziału
```

```
␣
↪===================================================================================

   Funkcja wyznacza zera niezależne oraz zera zależne.
   Zasada działania:

   1. Stworzenie listy list zawierającej informacje o pozycjach zer w każdym␣
↪wierszu macierzy matrix
   (jeśli zera nie ma w danym wierszu lista zawiera -1) oraz zmiennej best_sol␣
↪i best_sol_size odpowiednio
   przechowującej informacje o najlepszym rozwiązaniu i liczbie zer przez nie␣
↪wyznaczonej

   2. Wyznaczenie kombinacji tych pozycji zer
       przykład:
           wejście [[1, 2, 3], [-1], [0, 2]]
           wyjście [(1, -1, 0), (1, -1, 2), (2, -1, 0), (2, -1, 2), (3, -1,␣
↪0), (3, -1, 2)]

   3. Wyznaczenie najlepszego dopasowania:
           1. Przejście po wszystkich elementach:
               jeśli wartości w elemencie się powtarzają (z wyjątkiem -1):
                       zamień kolejne powtórzenia wartości na -1
               jeśli nie:
                       jeśli liczba niezależnych zer z danego elementu >␣
↪best_sol_size:
                           ustaw best_sol_size jako liczba niezależnych zer z␣
↪danego elementu (tj. rozmiar elementu - liczba wartości "-1")
                           ustaw best_sol jako ten element
                       jeśli nie:
                           wybierz następny element
               jeśli best_sol_size wynosi tyle co rozmiar macierzy:
                   zakończ
               jeśli nie:
                   sprawdź kolejny element

   4. Utwórz macierz z zerami zależnymi i niezależnymi an podstawie best_sol
                   0 - wartość różna od zera w macierzy matrix
                   1 - zero niezależne w macierzy matrix
                   2 - zero zależne w macierzy matrix

   5. Jeśli best_sol_size wynosi tyle co rozmiar macierzy matrix:
           zwróć informacje o przydziale, koszt i zakończ algorytm
      Jeśli nie:
           wywołaj kolejny etap
```

```
    :param matrix: przekształcona macierz z etapu I
    :param fi: aktualny koszt
    :return: informacja o przydziale i koszt lub macierz z I etapu, macierz z␣
↪zerami nie~/zależnymi, aktualny koszt
    """
    # etap 1
    lol: List[List[int]] = list()
    best_sol: Optional[Tuple[int]] = []
    best_sol_size: Optional[int] = 0
    size: int = matrix.shape[0]
    for row in matrix:
        lol.append([])
        for col in range(size):
            if row[col] == 0:
                lol[-1].append(col)
        if not len(lol[-1]):
            lol[-1].append(-1)

    # etap 2
    pred_sol: List[Tuple[int]] = list(product(*lol))

    # etap 3
    for elem in pred_sol:
        tmp = Counter(elem)
        tmp2 = False
        for i in range(size):
            if tmp[i] and tmp[i] > 1:
                tmp2 = True
                break
        if tmp2:
            elem = list(elem)
            for i in range(len(elem)):
                if elem[i] in elem[:i]:
                    elem[i] = -1
            elem = tuple(elem)
            tmp = Counter(elem)
        if (tmp[-1] and size - tmp[-1] > best_sol_size) or not tmp[-1]:
            best_sol_size = size - tmp[-1] if tmp[-1] else size
            best_sol = elem
        else:
            continue
        if best_sol_size == size:
            break

    # etap 4
    matrix_to_return: np.ndarray = np.zeros((size, size))
    for i in range(size):
```

```
        if lol[i][0] != -1:
            for el in lol[i]:
                matrix_to_return[i][el] = 2
    for i in range(size):
        if best_sol[i] != -1:
            matrix_to_return[i][best_sol[i]] = 1

    # etap 5
    if best_sol_size == size:
        fit: str = ""
        for i in range(size):
            fit += f"zadanie: {i} --> maszyna: {best_sol[i]}\n"
        print("\nMacierz zer niezależnych")
        print(matrix_to_return)
        print("\nDopasowanie:\n", fit, "\nKoszt:\n", str(fi))
    else:
        print("\nMacierz zer niezależnych")
        print(matrix_to_return)
        return cross_zeros_off(matrix, matrix_to_return, fi)
```

Komentarz Minimalna liczba zer niezależnych w macierzy zredukowanej jest równa jeden natomiast w optymistycznym przypadku maksymalna liczba zer niezależnych jest równa wymiarowi macierzy.

1.3 Krok 3 - Sprawdzenie liczby zer niezależnych

[11]:
```
#Zwraca minimalną listę wierszy i listę kolumn zawierających wszystkie zera w␣
↪macierzy = pokrycie wierzchołkowe
def cross_zeros_off(org:np.ndarray, A:np.ndarray, fi:float) -> Tuple[np.
↪ndarray, np.ndarray]:

    independent_zero = 1 #Oznaczenie zera niezależnego
    dependent_zero = 2 #Oznaczenie zera zależnego

    n = len(A) #Wymiar macierzy

    if n != len(A[0]): #Funkcja przyjmuje tylko macierze kwadratowe
        print("Macierz nie jest kwadratowa")
        return 0

    #True -> oznaczam wiersz lub kolumnę symbolem x
    B1 = np.zeros(n, bool) #Wektor zakreśleń wierszy
    B2 = np.zeros(n, bool) #Wektor zakreśleń kolumn

    new = 1
```

```
    #-----Oznaczanie symbolem x każdy wiersz nie posiadający zera niezależnego-----
    for i in range(n):
        independent_zero_exist = False
        for j in range(n):
            if A[i][j] == independent_zero:
                independent_zero_exist = True
        if independent_zero_exist == False:
            if B1[i] == False:
                new = 1
                B1[i] = True

    while new == 1: #Sprawdzenie czy w poprzedniej pętli dodano nowy element
        new = 0

        #-----Oznaczanie symbolem x każdą kolumnę posiadajęcą zero zależne w oznaczonym wierszu-----
        for i in range(n):
            for j in range(n):
                if A[i][j] == dependent_zero and B1[i]:
                    if B2[j] == False:
                        new = 1
                        B2[j] = True


        #-----Oznaczanie symbolem x każdy wiersz mający zero niezależne w oznakowanej kolumnie-----
        for i in range(n):
            for j in range(n):
                if A[i][j] == independent_zero and B2[j]:
                    if B1[i] == False:
                        new = 1
                        B1[i] = True

    #-----Pokrycie wierzchołkowe = wszystkie nieoznakowane wiersze i oznakowane kolumny-----
    row = np.logical_not(B1)
    column = B2

    print("\nOznaczone wiersze: ", row)
    print("Oznaczone kolumny: ", column)

    return step_4(org, fi, row, column)
```

Komentarz Minimalna liczba lini wykreślających wszystkie zera jest równa aktualnej liczbie zer niezależnych. Algorytm nie bierzę pod uwagę jak wiele zer jest w danym wierszu, sprawdza tylko czy istnieją.

1.4 Krok 4 - Próba powiększenia zbioru zer niezależnych

[12]:
```
def step_4(oryginal, fi, row, col):
    global counter
    counter += 1

    uncover_min = np.inf
    for i in range(oryginal.shape[0]):  # rzędy
        for j in range(oryginal.shape[1]):  # kolumny
            if not row[i] and not col[j] and oryginal[i, j] < uncover_min:
                uncover_min = oryginal[i, j]

    for i in range(oryginal.shape[0]):  # rzędy
        for j in range(oryginal.shape[1]):  # kolumny
            if not row[i] and not col[j]:
                oryginal[i, j] -= uncover_min
            if row[i] and col[j]:
                oryginal[i, j] += uncover_min


    print("\nmacierz po kroku 4")
    print(oryginal)
    return set_zeros(oryginal, fi+(uncover_min*counter))
```

Komentarz W sytuacji gdy liczba linii jest równa rozmiarowi macierzy to wiemy, że znaleziony zbiór zer niezależnych będzie miał tą samą wielskość. Każda iteracja algorytmu określana jest przez liczbę linii i kończy się gdy ilość ta jest równa rozmiarowi.

Metoda wyznaczania kolejnych zer jest zawsze skuteczna, ze względu na to, że przy każdej iteracji musi pojawić się kolejne zero. Liczba kolejnych wyznaczonych zer może być maksymalnie równa wymiarowi macierzy, jak zawarto w puntkach powyżej

1.5 Test algorytmu

[13]:
```
# Macierz kosztów

matrix = np.array([[5, 8, 9, 2, 1, 5, 8, 6, 4, 2],
                   [3, 4, 5, 8, 7, 5, 2, 1, 6, 5],
                   [9, 8, 7, 5, 9, 2, 3, 6, 5, 9],
                   [8, 7, 5, 6, 5, 7, 8, 9, 5, 6],
                   [2, 1, 4, 5, 7, 8, 6, 3, 2, 1],
                   [4, 7, 5, 8, 9, 6, 5, 2, 1, 4],
                   [7, 9, 5, 6, 3, 2, 1, 4, 5, 4],
                   [8, 2, 1, 5, 3, 2, 1, 2, 3, 6],
                   [5, 7, 5, 6, 2, 7, 9, 3, 4, 6],
```

```
                    [7, 5, 6, 4, 6, 7, 9, 2, 1, 7]])

print(matrix)
counter = 0
```

```
[[5 8 9 2 1 5 8 6 4 2]
 [3 4 5 8 7 5 2 1 6 5]
 [9 8 7 5 9 2 3 6 5 9]
 [8 7 5 6 5 7 8 9 5 6]
 [2 1 4 5 7 8 6 3 2 1]
 [4 7 5 8 9 6 5 2 1 4]
 [7 9 5 6 3 2 1 4 5 4]
 [8 2 1 5 3 2 1 2 3 6]
 [5 7 5 6 2 7 9 3 4 6]
 [7 5 6 4 6 7 9 2 1 7]]
```

[14]:

```
reduce_matrix(matrix)
```

```
Macierz ze zredukowanymi wierszami:
 [[4. 7. 8. 1. 0. 4. 7. 5. 3. 1.]
 [2. 3. 4. 7. 6. 4. 1. 0. 5. 4.]
 [7. 6. 5. 3. 7. 0. 1. 4. 3. 7.]
 [3. 2. 0. 1. 0. 2. 3. 4. 0. 1.]
 [1. 0. 3. 4. 6. 7. 5. 2. 1. 0.]
 [3. 6. 4. 7. 8. 5. 4. 1. 0. 3.]
 [6. 8. 4. 5. 2. 1. 0. 3. 4. 3.]
 [7. 1. 0. 4. 2. 1. 0. 1. 2. 5.]
 [3. 5. 3. 4. 0. 5. 7. 1. 2. 4.]
 [6. 4. 5. 3. 5. 6. 8. 1. 0. 6.]]
Koszt redukcji wierszy: 16

Zredukowana macierz:
 [[3. 7. 8. 0. 0. 4. 7. 5. 3. 1.]
 [1. 3. 4. 6. 6. 4. 1. 0. 5. 4.]
 [6. 6. 5. 2. 7. 0. 1. 4. 3. 7.]
 [2. 2. 0. 0. 0. 2. 3. 4. 0. 1.]
 [0. 0. 3. 3. 6. 7. 5. 2. 1. 0.]
 [2. 6. 4. 6. 8. 5. 4. 1. 0. 3.]
 [5. 8. 4. 4. 2. 1. 0. 3. 4. 3.]
 [6. 1. 0. 3. 2. 1. 0. 1. 2. 5.]
 [2. 5. 3. 3. 0. 5. 7. 1. 2. 4.]
 [5. 4. 5. 2. 5. 6. 8. 1. 0. 6.]]
Całkowity koszt redukcji macierzy: 18.0

Macierz zer niezależnych
[[0. 0. 0. 1. 2. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

```
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 2. 2. 0. 0. 0. 2. 0.]
 [1. 2. 0. 0. 0. 0. 0. 0. 0. 2.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 2. 0. 0. 0. 2. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 2. 0.]]

Oznaczone wiersze:  [False  True  True False  True False False False False
False]
Oznaczone kolumny:  [False False  True  True  True False  True False  True
False]

macierz po kroku 4
[[2. 6. 8. 0. 0. 3. 7. 4. 3. 0.]
 [1. 3. 5. 7. 7. 4. 2. 0. 6. 4.]
 [6. 6. 6. 3. 8. 0. 2. 4. 4. 7.]
 [1. 1. 0. 0. 0. 1. 3. 3. 0. 0.]
 [0. 0. 4. 4. 7. 7. 6. 2. 2. 0.]
 [1. 5. 4. 6. 8. 4. 4. 0. 0. 2.]
 [4. 7. 4. 4. 2. 0. 0. 2. 4. 2.]
 [5. 0. 0. 3. 2. 0. 0. 0. 2. 4.]
 [1. 4. 3. 3. 0. 4. 7. 0. 2. 3.]
 [4. 3. 5. 2. 5. 5. 8. 0. 0. 5.]]

Macierz zer niezależnych
[[0. 0. 0. 1. 2. 0. 0. 0. 0. 2.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 2. 2. 0. 0. 0. 2. 2.]
 [1. 2. 0. 0. 0. 0. 0. 0. 0. 2.]
 [0. 0. 0. 0. 0. 0. 0. 2. 2. 0.]
 [0. 0. 0. 0. 0. 2. 1. 0. 0. 0.]
 [0. 1. 2. 0. 0. 2. 2. 2. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 2. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 2. 1. 0.]]

Oznaczone wiersze:  [ True False  True  True  True False  True  True  True
False]
Oznaczone kolumny:  [False False False False False False False  True  True
False]

macierz po kroku 4
[[2. 6. 8. 0. 0. 3. 7. 5. 4. 0.]
 [0. 2. 4. 6. 6. 3. 1. 0. 6. 3.]
 [6. 6. 6. 3. 8. 0. 2. 5. 5. 7.]
 [1. 1. 0. 0. 0. 1. 3. 4. 1. 0.]
```

```
 [0. 0. 4. 4. 7. 7. 6. 3. 3. 0.]
 [0. 4. 3. 5. 7. 3. 3. 0. 0. 1.]
 [4. 7. 4. 4. 2. 0. 0. 3. 5. 2.]
 [5. 0. 0. 3. 2. 0. 0. 1. 3. 4.]
 [1. 4. 3. 3. 0. 4. 7. 1. 3. 3.]
 [3. 2. 4. 1. 4. 4. 7. 0. 0. 4.]]

Macierz zer niezależnych
[[0. 0. 0. 1. 2. 0. 0. 0. 0. 2.]
 [1. 0. 0. 0. 0. 0. 0. 2. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 2. 2. 0. 0. 0. 0. 2.]
 [2. 2. 0. 0. 0. 0. 0. 0. 0. 1.]
 [2. 0. 0. 0. 0. 0. 0. 1. 2. 0.]
 [0. 0. 0. 0. 0. 2. 1. 0. 0. 0.]
 [0. 1. 2. 0. 0. 2. 2. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 2. 1. 0.]]

Dopasowanie:
 zadanie: 0 --> maszyna: 3
zadanie: 1 --> maszyna: 0
zadanie: 2 --> maszyna: 5
zadanie: 3 --> maszyna: 2
zadanie: 4 --> maszyna: 9
zadanie: 5 --> maszyna: 7
zadanie: 6 --> maszyna: 6
zadanie: 7 --> maszyna: 1
zadanie: 8 --> maszyna: 4
zadanie: 9 --> maszyna: 8

Koszt:
 21.0
```

Komentarz Algorytm przydzielił zadania dla wszystkich maszyn oraz obliczył minimalny koszt dla przygotowanej macierzy kosztów. Jak widać dla tak zdefiniowanych danych zostały wykorzystane wszystkie kroki.