Sprawozdanie metody RSM i Safety Principles

Skład Zespołu:

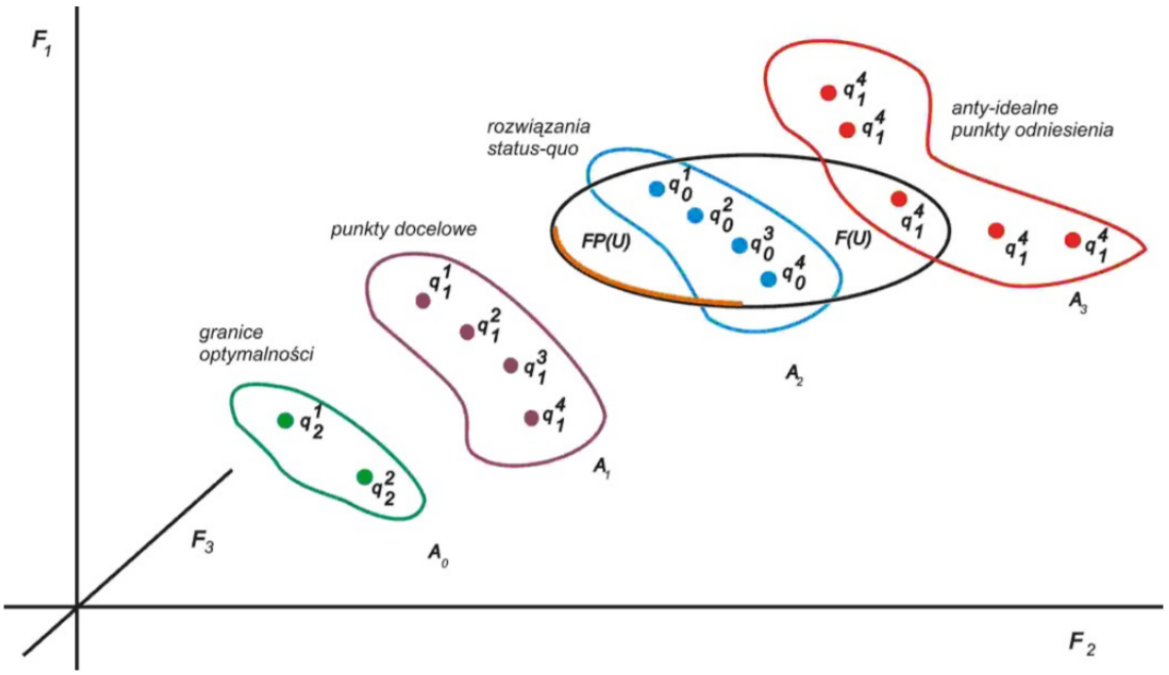
- Marcin Biela
- Jakub Sarata
- Michał Święciło
- Michał Spinczyk
- Konrad Kropornicki

Wstęp:

Metody punktów odniesienia (RSM) i Safety Principles (SP) są sposobami na poszukiwanie rozwiązań problemów optymalizacji wielokryterialnej. Są to zatem metody, które zastosować można przy problemach tworzenia rankingów np. wybór laptopa. Obie metody są do siebie podobne pod kilkoma względami. W obu algorytmach przed przystąpieniem do pracy należy określić odpowiednio 4 zbiory:

- A_0 - granica optymalności
- A_1 - punkty docelowe
- A_2 - rozwiązanie status-quo
- A_3 - anty-idealne punkty odniesienia

Zbiory te tworzymy w taki sposób, aby punkty wewnątrz nich były nieporównywalne i pkt w zbiorze o niższym indeksie dominowały nad pkt w zbiorze o indeksie wyższym.

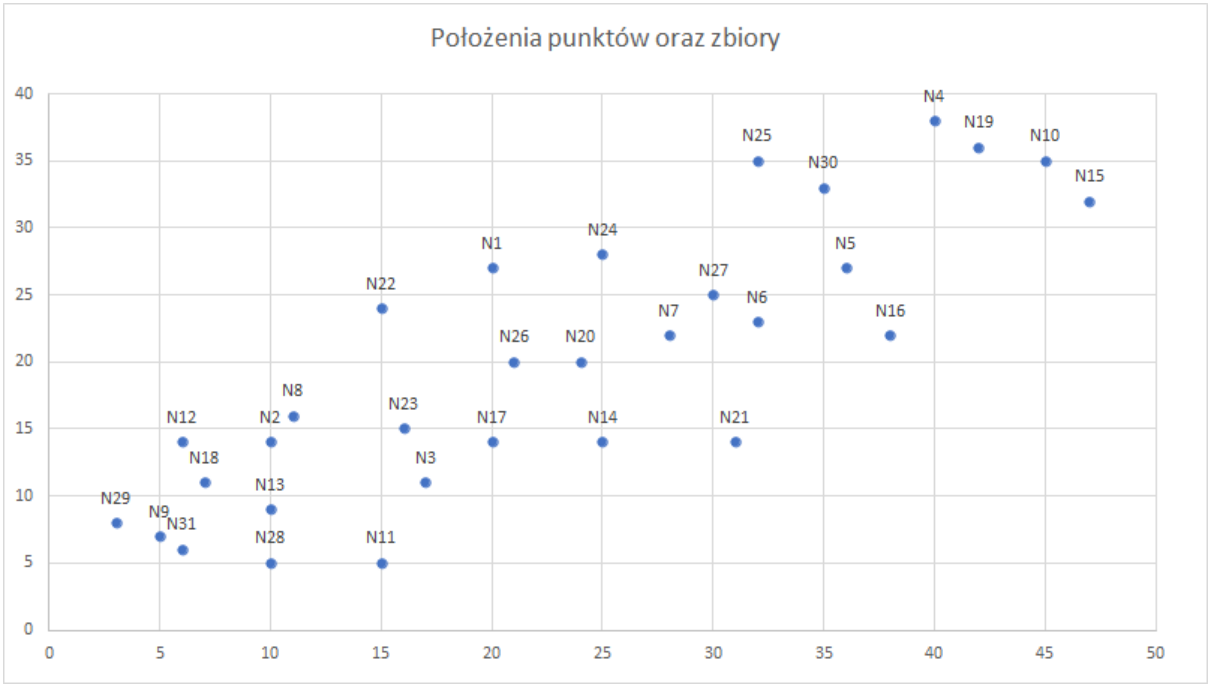


Jeżeli zbiory te zostały zdefiniowane poprawnie to pomiędzy zbiorem A_1 i A_2 pozostać powinny pkt nieprzypisane do żadnej z wymienionych wcześniej zbiorów. To właśnie do nich stosować będziemy metody RSM i SP. Będziemy chcieli stworzyć z tych pkt ranking pozwalający na uszeregowanie ich od najlepszego do najgorszego. Metody RSM i SP różnią się między sobą sposobem obliczania funkcji scoringowych.
W metodzie RSM tworzy się prostokąty, których jeden wierzchołek leży w zbiorze, a drugi w zbiorze A_2. Pole tak powstałego prostokąta jest współczynnikiem wagowym. Tworzymy tyle takich prostokątów ile jest możliwe. Następnie dla każdego pkt u (znajdującego się pomiędzy zbiorami A_1 i A_2) obliczamy odległości od danch pkt będących wierzchołkami prostokąta i przemnażamy ją przez współczynnik wagowy danego prostokąta. Tak czynimy kolejno dla każdego prostokąta, a współczynnik skoringowy jest sumą wszystkich takich mnożeń. Jeżeli dany pkt nie zawiera się w prostokącie to współczynnik wagowy przyjmujemy 0. Współczynniki wagowe dla danego pkt u muszą być znormalizowane tak, aby ich suma wynosiła 1.

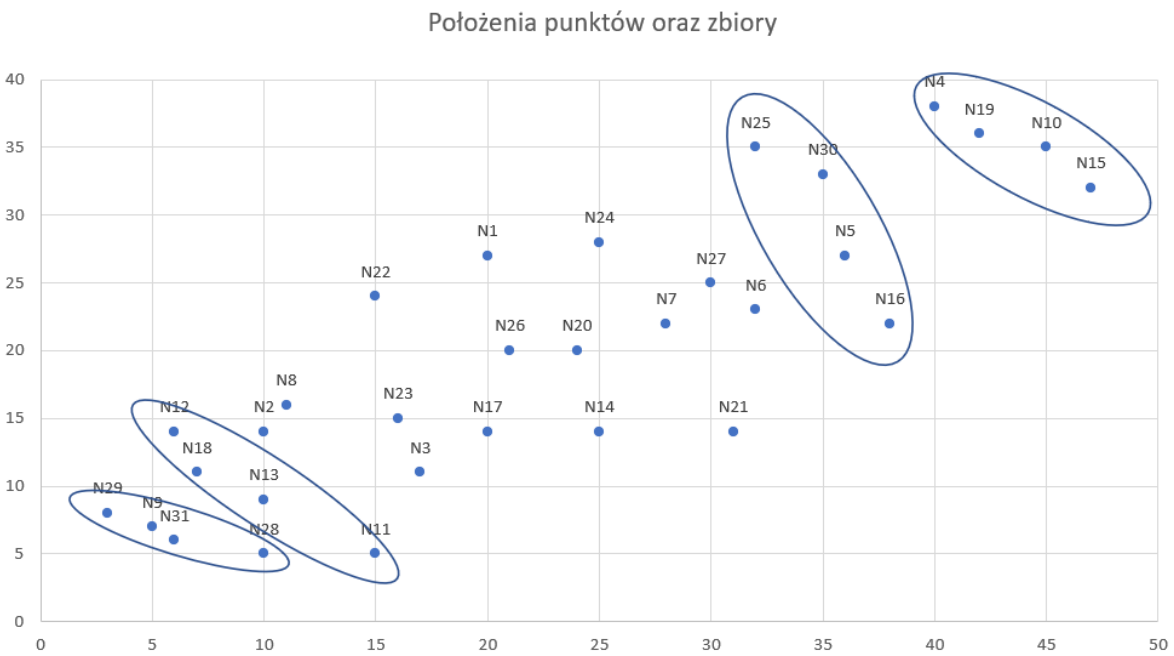
W metodzie SP działamy podobnie z tą różnicą, że zamiast prostokątów tworzymy krzywe Woronoja. Następnie rozważany pkt u rzutujemy na każdą z krzywych, a współczynnik skoringowy jest sumą współczynników rzutów na wszystkie krzywe. Również i tutaj współczynniki na krzywych Woronoja normalizujemy w taki sposób, aby dla pkt ze zbioru A_1 wartość wynosiła 0, a dla pkt ze zbioru A_2 1.

Zbiór danych:

Na potrzeby projektu grupa wyselekcjonowała 30 elementowy zbiór decyzji U oznaczone indeksami (N_1, N_2,…, N_i, i=30) o dwóch kryteriach X i Y należących do zbioru liczb całkowitych (X, Y ∈ Z). Decyzje te wybrano odpowiednio tak, aby można było stworzyć odpowiednie zbiory punktów odniesienia określając poziomy odniesienia dla poszczególnych kryteriów. Poziomy pozwalają na określenie pożądalności poszczególnych rezultatów np. punkty aspiracji odpowiadające pożądanym rezultatom oraz rezerwacji, odpowiadające najgorszym akceptowalnym rezultatom.



Wykres zbioru decyzji.



Wykres zbioru decyzji z naniesionymi zbiorami punktów odniesienia.

Napisane algorytmy:

- **Metoda punktów odniesienia (RSM)**

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
from typing import Dict, Tuple, List, Union, Optional

# Wczytywanie zbioru punktów
df = pd.read_excel("zbior_punktow.xlsx", 'Arkusz2')

# Przerobienie zbioru punktów z postaci pd.DataFrame to dict: key:
'NX', value: (x, y),  1 <= X <=30
dct = dict()
pkt_nazwa = list(df['U'])
pkt_x = list(df['x'])
pkt_y = list(df['y'])
for i in range(len(pkt_x)):
    dct[pkt_nazwa[i]] = (pkt_x[i], pkt_y[i])
print(dct)

df2 = pd.read_excel("zbior_punktow.xlsx", 'Arkusz3')


# Wyodrębnienie odpowiednich punktów do poszczególnych klas A0, A1, A2,
A3
# A0 - granice optymalności
# A1 - punkty docelowe
# A2 - punkty status - quo
# A3 - anty idealne - punkty odniesienia


A0_idx = df2.A0.to_list()
A1_idx = df2.A1.to_list()
A2_idx = df2.A2.to_list()
A3_idx = df2.A3.to_list()

A0 = []
A1 = []
A2 = []
A3 = []

for i in range(len(A0_idx)):
    A0.append(dct[A0_idx[i]])
    A1.append(dct[A1_idx[i]])
    A2.append(dct[A2_idx[i]])
    A3.append(dct[A3_idx[i]])

def check(u: Tuple, point_A_1: Tuple, point_A_2: Tuple) -> bool:
```

```
    """
    Sprawdzanie czy dana dany punkt znajduje się w prostokącie
stworzonym przez punktu ze
    zbioru A1 i A2

    params: -u: Tuple(int, int)
            -A1_point: Tuple(int, int),
            -A2_point: Tuple(int, int),

    Return: bool
    """

    x_min, y_min = point_A_1
    x_max, y_max = point_A_2
    x, y = u
    if ((x >= x_min) and (x <= x_max)) and ((y >= y_min) and (y <=
y_max)):
        return True
    else:
        return False

def oblicz_pole(A1_point: Tuple, A2_point: Tuple, u: Tuple) -> Tuple:
    """
    Funkcja obliczająca pole pomiędzy A1_point, A2_point i sprawdza czy
punkt ze zbioru status-quo znajduje się w
    obrębie tego prostokąta.


    params: -A1_point: Tuple(int, int),
            -A2_point: Tuple(int, int),
            -u: Tuple(int, int)

    Return: Tuple(Optional[Union[float, int], str, str])
    """

    x_1, y_1 = A1_point
    x_2, y_2 = A2_point
    p1 = list(dct.keys())[list(dct.values()).index((x_1, y_1))]
    p2 = list(dct.keys())[list(dct.values()).index((x_2, y_2))]
    if check(u,A1_point,A2_point):
        return (abs(x_1 - x_2)*abs(y_1 - y_2), p1, p2)
    else:
        return (0, p1, p2)
```

```python
def wagi(A1: Tuple ,A2: Tuple,u: Tuple) -> List:
    """
    Funkcja obliczająca wagi na podstawie punktów A1, A2, u
    Wewnętrznie sprawdzany jest warunek czy dany punkt u znajduje się w prostokącie A1, A2

    params: -A1: List[Tuple(int, int)],
            -A2: List[Tuple(int, int)],
            -u: Tuple(int, int)

    Return: List[Tuple(float, str, str)]
    """
    suma = 0
    pola = []
    wagi_ = []

    for A1_point in A1:
        for A2_point in A2:
            x_1, y_1 = A1_point
            x_2, y_2 = A2_point
            p1 = list(dct.keys())[list(dct.values()).index((x_1, y_1))]
            p2 = list(dct.keys())[list(dct.values()).index((x_2, y_2))]
            pole = oblicz_pole(A1_point,A2_point,u)[0]
            suma += pole
            pola.append((pole,p1,p2))

    for pole_ in pola:
        pole,p1,p2 = pole_
        if suma != 0:
            waga = pole/suma
        else:
            waga = 0
        wagi_.append((waga,p1,p2))

    return wagi_


def check_if_weight_sum_to_1(A1, A2, u):
    """
    Funkcja sprawdzająca czy obliczone wagi z danego punktu u sumują się do 1
    """
```

```
    x = wagi(A1,A2,u)
    suma = 0
    for i in x:
        suma += i[0]
    return suma

suma = check_if_weight_sum_to_1(A1, A2, dct['N1'])
print("Sprawdzenie czy suma ", suma)

# Zbiór punktów status-quo
B0 = []

A1xmin = min([point[0] for point in A1])
A2xmax = max([point[0] for point in A2])

# print('A1xmin ', A1xmin)
# print('A1xmax ', A2xmax)

for point_name in dct.keys():
    if A1xmin < dct[point_name][0] < A2xmax:
        if (dct[point_name] not in A1) and (dct[point_name] not in A2)
and (dct[point_name] not in A0) and (dct[point_name] not in A3):
            B0.append(dct[point_name])

plt.figure()
plt.scatter([point[0] for point in B0], [point[1] for point in B0])
plt.show()
print(B0)

# print(check(dct['N21'],dct['N11'],dct['N16']))
def distance(u: Tuple, A: Tuple) -> float:
    """
    Funkcja obliczająca dystans z punktu u do A
    params: u - Tuple(int, int)
            A - Tuple(int, int)

    return: float
    """
    x_A, y_A = A
    x, y = u
    d = np.sqrt((x-x_A)**2+(y-y_A)**2)
    return d
```

```python
def skoring(u: Tuple, A1: List[Tuple], A2: List[Tuple]) -> Tuple:
    """
    Funkcja wyliczajaca wartość funkcji skoringowej dla danego
    punktu u ze zbioru status quo

    params:
    u - Tuple(int, int)
    A1 - List[Tuple(int, int)]
    A2 - List[Tuple(int, int)]

    """
    wagi_ = wagi(A1,A2,u)
    F = 0
    for waga, A1_point, A2_point in wagi_:
            d_idealny = distance(u,dct[A1_point])
            d_anty = distance(u,dct[A2_point])
            f = d_anty/(d_anty + d_idealny)
            F += waga * f
    x,y = u
    u_ = list(dct.keys())[list(dct.values()).index((x, y))]
    return (F,u_)

# Poniższa część kodu odpowiada za tworzenie rankingu
# ranking jest zwracany w postaci List[Tuple(float, str')]
# listy punktów od najlepszego do najgorszego

ranking = []

for point in B0:
    ranking.append(skoring(point,A1,A2))

ranking = sorted(ranking, key = lambda tup: tup[0], reverse=True)

ranking_to_plot = [f'{i+1}: {ranking[i]}' for i in range(len(ranking))]

print("Ranking:\n")
for elem in ranking_to_plot:
    print(elem)
```

- **Safety Principles (SP)**

```
# Zaimportowanie niezbędnych bibliotek
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from typing import Dict, Optional, List, Union
from numpy.linalg import norm


# Wczytywanie zbioru punktów
df = pd.read_excel("zbior_punktow.xlsx", 'Arkusz2')


# Przerobienie zbioru punktów z postaci pd.DataFrame to dict: key:
'NX', value: (x, y),  1 <= X <=30
dct = dict()
pkt_nazwa = list(df['U'])
pkt_x = list(df['x'])
pkt_y = list(df['y'])
for i in range(len(pkt_x)):
    dct[pkt_nazwa[i]] = (pkt_x[i], pkt_y[i])


# Wyodrębnienie odpowiednich punktów do poszczególnych klas A0, A1, A2,
A3
```

```
# A0 - granice optymalności
# A1 - punkty docelowe
# A2 - punkty status - quo
# A3 - anty idealne - punkty odniesienia
df2 = pd.read_excel("zbior_punktow.xlsx", 'Arkusz3')

A0_idx = df2.A0.to_list()
A1_idx = df2.A1.to_list()
A2_idx = df2.A2.to_list()
A3_idx = df2.A3.to_list()

A0 = []
A1 = []
A2 = []
A3 = []

for i in range(len(A0_idx)):
    A0.append(dct[A0_idx[i]])
    A1.append(dct[A1_idx[i]])
    A2.append(dct[A2_idx[i]])
    A3.append(dct[A3_idx[i]])


# Zbiór punktów status-quo
B0 = []

A1xmin = min([point[0] for point in A1])
A2xmax = max([point[0] for point in A2])

for point_name in dct.keys():
    if A1xmin < dct[point_name][0] < A2xmax:
        if (dct[point_name] not in A1) and (dct[point_name] not in A2)
and (dct[point_name] not in A0) and (dct[point_name] not in A3):
            B0.append(dct[point_name])
def plot_points(zbior: Optional[Union[Dict, List]], dct: Dict=dct):
    """
    Funkcja tworząca wykres punktów z danego zbioro (scatter plot).

    params: -zbior: Union[Dict, List]

    Return: None
    """
```

```
    if isinstance(zbior, Dict):
        fig, ax = plt.subplots()
        fig.set_size_inches((20, 15))
        ax.set_ylim((0, 40))
        ax.set_xlim((0, 50))

        ax.scatter([zbior[point][0] for point in zbior.keys()],
[zbior[point][1] for point in zbior.keys()])

        for key, coords in zbior.items():
            ax.annotate(key, coords)

    if isinstance(zbior, List):
        fig, ax = plt.subplots()
        fig.set_size_inches((20, 15))
        ax.set_ylim((0, 40))
        ax.set_xlim((0, 50))

        ax.scatter([point[0] for point in zbior], [point[1] for point
in zbior])

        # dct =
list(zbior.keys())[list(zbior.values()).index((point[0], point[1]))]


        for coords in zbior:
            key = list(dct.keys())[list(dct.values()).index((coords[0],
coords[1]))]
            ax.annotate(key, coords)


def krzywa_woronoya(u, A1_point, A2_point):
    """
    Funkcja tworząca wykres krzywej Woronoya wraz z punktem u.
    params: -u: Tuple(int, int)
            -A1_point: Tuple(int, int)
            -A2_point: Tuple(int, int)

    Return: None
    """
    fig, ax = plt.subplots()
    fig.set_size_inches((10, 7))
    ax.set_ylim((0, 40))
```

```
    ax.set_xlim((0, 50))
    _, f1, f2, _ = stworz_krzywa(A1_point, A2_point)
    zbior = [u, A1_point, f1, f2, A2_point]

    zbior_to_text = [u, A1_point, A2_point]
    ax.scatter([point[0] for point in zbior], [point[1] for point in
zbior])

    # dct = list(zbior.keys())[list(zbior.values()).index((point[0],
point[1]))]
    ax.plot([zbior[i][0] for i in range(1, len(zbior))], [zbior[i][1]
for i in range(1, len(zbior))])
    for coords in zbior_to_text:
        key = list(dct.keys())[list(dct.values()).index((coords[0],
coords[1]))]
        ax.annotate('  '+key+f' {coords}', coords)
    ax.annotate(f'f1: {f1}', f1)
    ax.annotate(f'f2: {f2}', f2)
def oblicz_d(A1_point,A2_point):
    """
    Funkcja obliczająca zbiór d wraz z uporządkowaniem elementów
zbioru.
    params: -A1_point: Tuple(int, int)
            -A2_point: Tuple(int, int)

    Return: float
    """
    x_A1, y_A1 = A1_point
    x_A2, y_A2 = A2_point
    d1 = (x_A2 - x_A1)/2
    d2 = (y_A2 - y_A1)/2
    # print(d1,d2)
    # if d1 > d2:
    #     d1, d2 = d2, d1
    return min(d1,d2)

def stworz_krzywa(A1_point,A2_point):
    """
    Funkcja wyznaczająca punktów w których krzywa się łamie.
    params: -A1_point: Tuple(int, int)
            -A2_point: Tuple(int, int)
```

```
    Return: (Tuple(int, int),Tuple(int, int),Tuple(int, int),Tuple(int,
int))
    """
    d = oblicz_d(A1_point,A2_point)
    x_A1, y_A1 = A1_point
    x_A2, y_A2 = A2_point
    f1 = (x_A1+d,y_A1+d)
    f2 = (x_A2-d,y_A2-d)
    return (A1_point,f1,f2,A2_point)

def odleglosc(u, A1_odc, A2_odc):
    """
    Funkcja obliczające odleglosc punktu od prostej zawierajacej
odcinek, nie uwzględnia przypadku jeżeli
    punkt nie zawiera się w odcink

    params: -u: Tuple(int, int)
            -A1_odc: Tuple(int, int)
            -A2_odc: Tuple(int, int)

    Return: float
    """
    #  Funkcja obliczające odleglosc punktu od prostej
    # zawierajacej odcinek nie uwzględnia przypadku jeżeli
    # punkt nie zawiera się w odcinku
    u = np.asarray(u)
    A1_odc = np.asarray(A1_odc)
    A2_odc = np.asarray(A2_odc)

    d = norm(np.cross(A2_odc-A1_odc, A1_odc-u))/norm(A2_odc-A1_odc)
    return d

def oblicz_dlugosc_odcinka(point_1, point_2):
    """
    Funkcja obliczające odleglosc pomiędzy dwoma punktami.

    params: -point_1: Tuple(int, int)
            -point_2: Tuple(int, int)

    Return: float
    """
    x_1, y_1 = point_1
    x_2, y_2 = point_2
```

```
    return ((x_2-x_1)**2 + (y_2 - y_1)**2)**(1/2)

def sprawdz_czy_punkt_u_w_odcinku_AB(u, point_1, point_2):
    """
    Funkcja sprawdzająca czy punkt u znajduje się pomiędzy odcinkiem
AB.

    params: -point_1: Tuple(int, int)
            -point_2: Tuple(int, int)
            -u: Tuple(int, int)

    Return: bool

    https://matematyka.pl/viewtopic.php?t=372091
    Znając współrzędne punktu P i końców odcinka AB możesz policzyć
odległości między nimi. Jeśli te długości tworzą trójkąt rozwartokątny
i AB nie
    jest najdłuższym bokiem trójkąta to rzut punktu na prostą
zawierającą odcinek jest poza nim.
    Kąt leżący naprzeciw boku a jest rozwarty w trójkącie o bokach a,
b, c gdy
    """
    odc_u_point_1 = oblicz_dlugosc_odcinka(u, point_1) # c
    odc_u_point_2 = oblicz_dlugosc_odcinka(u, point_2) # b
    odc_point_1_point_2 = oblicz_dlugosc_odcinka(point_1, point_2) # a

    # Czy punkt u znajduje się w odcinku AB
    result = False
    lst_odcinek = [odc_u_point_1, odc_point_1_point_2, odc_u_point_2]

    a_idx = lst_odcinek.index(max(lst_odcinek))
    a = lst_odcinek[a_idx]

    czy_trojkat_rozw = False
    if odc_point_1_point_2+odc_u_point_2 > odc_u_point_1 and
odc_point_1_point_2+odc_u_point_1>odc_u_point_2 and
odc_u_point_2+odc_u_point_1>odc_point_1_point_2:
        if odc_point_1_point_2**2+odc_u_point_2**2<odc_u_point_1**2 or
odc_point_1_point_2**2+odc_u_point_1**2 < odc_u_point_2**2 or
odc_u_point_2**2+odc_u_point_1**2<odc_point_1_point_2**2:
            czy_trojkat_rozw = True
        else:
            czy_trojkat_rozw = False
```

```
    if czy_trojkat_rozw and odc_point_1_point_2 != a:
        result = False

    else:
        result = True

    return result

def point_on_line(u, a, b):
    """
    Funkcja tworząca rzut punktu na prostą.

    params: -u: Tuple(int, int)
            -a: Tuple(int, int)
            -b: Tuple(int, int)

    Return: Tuple(int, int)
    """
    au = u - a
    ab = b - a
    t = np.dot(au, ab) / np.dot(ab, ab)
    # if you need the the closest point belonging to the segment
    t = max(0, min(1, t))
    # print('ab', ab, 'au', au, 't', t)
    result = a + t * ab
    return result


def oblicz_odleglosc(u, A1_point, A2_point):
    """
    Funkcja obliczająca odległości pomiędzy punktem a wszystkimi
prostymi.

    params: -u: Tuple(int, int)
            -A1_point: Tuple(int, int)
            -A2_point: Tuple(int, int)

    Return: Tuple(float, int)
    """
    A1_point, f1, f2, A2_point = stworz_krzywa(A1_point,A2_point)
    # print(A1_point, f1, f2, A2_point)
    lista = [A1_point, f1, f2, A2_point]
```

```
    x_u, y_u = u

    #
https://stackoverflow.com/questions/39840030/distance-between-point-and
-a-line-from-two-points

    # u = np.asarray(u)
    # f1 = np.asarray(f1)
    # f2 = np.asarray(f2)
    # A1_point = np.asarray(A1_point)
    # A2_point = np.asarray(A2_point)
    d1 = np.Inf
    d2 = np.Inf
    d3 = np.Inf
    # Odleglosc u od prostej A1_point, f1
    # d1 = norm(np.cross(f1-A1_point, A1_point-u))/norm(A1_point-u)
    # print("A1_point-f1", sprawdz_czy_punkt_u_w_odcinku_AB(u,
A1_point, f1))
    if sprawdz_czy_punkt_u_w_odcinku_AB(u, A1_point, f1):
        d1 = odleglosc(u, A1_point, f1)
    # Odleglosc u od prostej f1, f2
    # d2 = norm(np.cross(f2-f1, f1-u))/norm(f1-u)
    if sprawdz_czy_punkt_u_w_odcinku_AB(u, f1, f2):
        d2 = odleglosc(u, f1, f2)
    # print("f1-f2", sprawdz_czy_punkt_u_w_odcinku_AB(u, f1, f2))
    # Odleglosc u od prostej f2, A2_point
    # d3 = norm(np.cross(A2_point-f2, f2-u))/norm(f2-u)
    if sprawdz_czy_punkt_u_w_odcinku_AB(u, f2, A2_point):
        d3 = odleglosc(u, f2, A2_point)
    # print("f2-A2_point", sprawdz_czy_punkt_u_w_odcinku_AB(u, f2,
A2_point))

    lst_of_d = [d1, d2, d3]
    return min(d1, d2, d3), lst_of_d.index(min(lst_of_d))

def oblicz_wart_y(u, A1_point, A2_point):
    """
    Funkcja obliczająca wartość y.

    params: -u: Tuple(int, int)
            -A1_point: Tuple(int, int)
            -A2_point: Tuple(int, int)
```

```
    Return: Tuple(float, int)
    """
    u = np.asarray(u)
    A1_point = np.asarray(A1_point)
    A2_point = np.asarray(A2_point)

    x = point_on_line(u, A1_point, A2_point)
    return x

# print(oblicz_d(dct["N22"],dct['N12']))
# print(oblicz_odleglosc(dct['N22'],dct['N12'],dct['N5']))

# def odleglosc(point_1,point_2):
#     x_1, y_1 = point_1
#     x_2, y_2 = point_2
#     return ((x_2-x_1)**2 + (y_2 - y_1)**2)**(1/2)

def oblicz_wspolczynnik_skoringowy(u,A1_point,A2_point):
    """
    Funkcja obliczająca współczynnik skoringowy.

    params: -u: Tuple(int, int)
            -A1_point: Tuple(int, int)
            -A2_point: Tuple(int, int)

    Return: float
    """
    # odle, idx, f1, f2 = oblicz_odleglosc(u,A1_point,A2_point)
    # odleglosci =
[odleglosc(A1_point,f1),odleglosc(f1,f2),odleglosc(f2,A2_point)]
    # suma = odleglosci[0] + odleglosci[1] + odleglosci[2]
    y_max = A2_point[1]
    y_min = A1_point[0]
    x = oblicz_wart_y(u, A1_point, A2_point)
    return (x[1]-y_min)/(y_max-y_min)
    # suma_1 = suma/suma
    # # suma = 1
    # waga = odleglosci[idx]/suma
    # wsp_skoringowy = odle * waga
    # return wsp_skoringowy
```

```
# Obliczenie współczynnika skoringowego dla każdego punktu oraz
stworzenie rankingu.
ranking = []

for point in B0:
    suma = 0
    for A1_point in A1:
        for A2_point in A2:
            # suma += 1
            suma +=
oblicz_wspolczynnik_skoringowy(point,A1_point,A2_point)
            # krzywa_woronoya(point, A1_point, A2_point)
            # print("wspol",
oblicz_wspolczynnik_skoringowy(point,A1_point,A2_point))
    point = list(dct.keys())[list(dct.values()).index((point[0],
point[1]))]
    ranking.append((point,suma))


# Posortowanie wartości od najmniejszej do największej.
ranking = sorted(ranking, key = lambda tup: tup[1], reverse=False)
# print(oblicz_wspolczynnik_skoringowy())
# print(ranking)
# X = oblicz_odleglosc(dct['N24'], dct['N11'], dct['N25'])
# print(X)
# krzywa_woronoya(dct['N24'], dct['N11'], dct['N25'])
ranking_to_plot = [f'{i+1}: {ranking[i]}' for i in range(len(ranking))]
# print(ranking_to_plot)
print("Ranking:\n")
for elem in ranking_to_plot:
    print(elem)
```

Uzyskane wyniki:

- **Metoda punktów odniesienia (RSM)**

```
Ranking:

1: (0.872009647750395, 'N2')
2: (0.8175453206812621, 'N8')
3: (0.7724528316196977, 'N3')
4: (0.7210594696641758, 'N23')
5: (0.6499078205493714, 'N17')
6: (0.5723620412317619, 'N22')
7: (0.540082620356766, 'N14')
8: (0.5272976015490336, 'N26')
9: (0.4613838641660679, 'N20')
10: (0.429849908005132, 'N21')
11: (0.42810143189080707, 'N1')
12: (0.34129465521924957, 'N7')
13: (0.3016601097574758, 'N24')
14: (0.26281155907358733, 'N6')
15: (0.2519370440578506, 'N27')
```

- **Safety Principles (SP)**

```
Ranking:

1: ('N2', 0.446118895265293 47)
2: ('N8', 1.5163168593959648)
3: ('N3', 2.266589548802626)
4: ('N23', 3.1978910300992616)
5: ('N17', 4.476433718480989)
6: ('N22', 5.797108715918653)
7: ('N14', 6.491591130088836)
8: ('N26', 6.880964646229866)
9: ('N20', 8.090059093194572)
10: ('N1', 8.813015850240154)
11: ('N21', 8.909780024018252)
12: ('N7', 10.369351504289954)
13: ('N24', 11.161756502752553)
14: ('N27', 12.176164191646748)
15: ('N6', 12.315060674480785)
```

Porównując rankingi ze sobą możemy zauważyć, że są one niemal identyczne. Jedyna różnica to zamiana dwóch par punktów (N1-N21 i N27-N6), która jest rezultatem odmiennego sposobu obliczania rankingu.

Literatura i źródła wiedzy:

1. Dokumentacja oraz instrukcje zawarte na kursie przedmiotu ze strony UPEL.

2. Skulimowski A.M.J. (1996) Decision Support Systems Based on Reference Sets. Wydawnictwa AGH, Monografie, Nr 40, s. 167

3. Skulimowski A.M.J. (2019). Selected methods, applications, and challenges of multicriteria optimization. Seria Monografie, t.19, Komitet Automatyki i Robotyki Polskiej Akademii Nauk, Wydawnictwa AGH, s. 380

4. Andrzej M.J. SKULIMOWSKI (1986). Foreseen Utility in Multi-Stage Multicriteria Optimization. Seminar on Nonconventional Problems of Optimization, Warszawa, May 9-11, 1984. W: J. Kacprzyk (ed.), Proceedings, Part III, s. 365-386.

Rozkład obowiązków przy projekcie:

	Wykonawcy
Metoda RSM	- Marcin Biela - Jakub Sarata - Michał Spinczyk
Metoda SP	- Michał Święciło - Konrad Kropornicki - Marcin Biela
Sprawozdanie	- Jakub Sarata - Michał Spinczyk - Michał Święciło - Konrad Kropornicki