Звіт

Свою програму пошуку найкоротшого шляху по поверхні ми грунтували на алгоритмі Левіта, який є деяким покращенням алгоритму Дейкстри.

У програмі реалізований пошук найкоротшого шляху через алгоритм Левіта.

Алгоритм:

1)Створити масив (у нашому випадку список)для збереження шляхів до точок;

2) На початку ініціалізувати його зі значеннями infinity(у програмі таке значення позначає None), а відстань до початкової точки позначити 0.

3)Створити масив(список) ‘предків’ точок у найкоротших шляхах до них, з початковими значеннями відповідними точками для кожної позиції.

4)Створити список(в оригіналі алгоритму чергу) для збереження вершин, до яких ми обраховуємо відстань в даний момент.(нехай тут м1)

5)Створити список для збереження, ‘станів’ кожної вершини, для відслідкування в якій множині, категорії вершин вона перебуває(1 – вершини, до яких ми обраховуємо відстань зараз, 2 – вершини, до яких відстань ще не обраховувалась (нехай тут множина м2), 0 – вершини, до яких відстань уже обрахована(нехай тут множина м0), але ,можливо, ще не остаточно)

(У програмі є ще отримання розмірностей ландшафту.)

5)Ітерувати через м1, для кожного елементу, переміщувати його в м0(оновлювати значення в списку станів), знаходити суміжні точки, ітерувати через них і виконувати наступні дії:

А)Обраховувати відстань ребра з цією суміжною точкою;

Б)Якщо ця суміжна точка у множині м2,

то присвоїти значення знайденої відстані, якщо відстань до точки з м1 None або значення суми відстані до цієї точки і довжини ребра між даними точками.Перемістити у м1. Повторити для всіх суміжних точок.

В)Якщо точка в м1, оновити її значення мінімальним значенням серед старої обрахованої відстані і нової.

Г)Якщо точка у м0, оновити її значення мінімальним значенням серед старої обрахованої відстані і нової.

(Якщо значення ‘предка’ для точки з м1, яку ми розглядаємо, співпадає зі значенням суміжної точки, ребро з якою ми обраховуємо, існує варіант розвитку подій при якому пройти з такої суміжної точки туди назад до точки з м1 швидше ніж зі ‘старого предка’. Тоді програма шлях від предка зациклюється і це унеможливлює пошук необхідного шляху. Для цього у програмі передбачена додаткова умова.)

6)Результат конвертується у одинарні індекси і повертається як список.

```
#Getting size.
size = landscape.shape

#Creating empty list distances for storing
#minimal distances from the given point to a point in the grid
distances = [[None]*size[1]]*size[0]

#Creating a list ancestors for storing
#a point to cross to get to the point in the grid through the shortest path.
ancestors = [[(index, jindex) for jindex in
              range(size[1])] for index in range(size[0])]

#Setting up distance to a starting point as zero.
distances[pointa[0]][pointa[1]] = 0

#Creating a list for storing inf about in what array point is included.
points_ids = [[2 for jindex in
              range(size[1])] for index in range(size[0])]
points_ids[pointa[0]][pointa[1]] = 0

#Creating a list-queue for storing points which are under evaluation.
under_computation_distances = [pointa]
```

```
    #Iterating through the list of adjacent points.
    for end in ribs_finishes:
        end_cord0 = end[0]
        end_cord1 = end[1]
        #Getting height of an adjacent point and
        #computing distance with the evaluated one.
        height2 = landscape[end_cord0][end_cord1]
        distance = compute_distance(height1, height2, step)

        #If the adjacent point hasn't been evaluated with
        #any other point, compare a stored distance to
        #the other point and the distance with the adjacent one
        #and storing the minimal.
        if points_ids[end_cord0][end_cord1] == 2:
            try:
                new_distance = distances[element_cord0][element_cord1] + distance
                distances[end_cord0][end_cord1] = get_minimum(distances[end_cord0]
                                                              [end_cord1],
                                                              new_distance,
                                                              ancestors,
                                                              element)
            except ValueError:
                distances[end_cord0][end_cord1] = get_minimum(distances[end_cord0]
                                                              [end_cord1],
                                                              distance,
                                                              ancestors,
                                                              element)
            #Moving the adjacent point to the set of evaluated points.
            points_ids[end_cord0][end_cord1] = 1
            under_computation_distances.append(end)
            #Changing an ancestor for the adjacent point.
            ancestors[end_cord0][end_cord1] = element
```

```
#If the adjacent point is been evaluated,
#choose a minimum distance to it.
elif points_ids[end_cord0][end_cord1] == 1:
    distances[end_cord0][end_cord1] = get_minimum(distances[end_cord0][end_cord1],
                                  distances[element_cord0][element_cord1]
                                  + distance, ancestors, element)


#If the adjacent point has been previously
#evaluated and the rib with the the other point
#hasn't been evaluated yet, running next block.
elif points_ids[end_cord0][end_cord1] == 0:
    #Getitng an old distance and computing a new one.
    old_distance = distances[end_cord0][end_cord1]
    new_distance = distances[element_cord0][element_cord1] \
                                  + distance

    #If the adjacent point and an ancestor of the point we are
    #evaluating mathces, we can get a cycled path between the points.
    #This condition prevents it.
    if new_distance < old_distance and ancestors[element_cord0][element_cord1] != end:
        distances[end_cord0][end_cord1] = new_distance
        ancestors[end_cord0][end_cord1] = element
        points_ids[end_cord0][end_cord1] = 1
        under_computation_distances.insert(0, end)
```

```
    #Computing a path
    path = np.array([pointb])
    to_go_point = ancestors[pointb[0]][pointb[1]]

    #Creating a last visited point for storing
    #a point from which to move back to the pointa
    #on the current iteration.

    last_visited = to_go_point
    path = np.insert(path, 0, [to_go_point],  axis = 0)
    while tuple(to_go_point) != pointa:
        #Finding a next point in the path.
        to_go_point = ancestors[last_visited[0]][last_visited[1]]
        path = np.insert(path, 0, [to_go_point],  axis = 0)
        #Changing the last visited point.
        last_visited = to_go_point

    #Returning the result value.
    return path

if __name__ == '__main__':
    landscape_input, pointA_input, pointB_input, step_input = input(), input(), input(), input()
    start = time.time()
    path_output = levits_algortihm(landscape_input, pointA_input, pointB_input, step_input)
    print('Path:')
    print()
    print(path_output)
    print()
    print(f'Runtime: {time.time() - start}')
    print()
    with open('results__.txt', 'w', encoding='utf-8') as results_file:
        for step_forward in path_output:
            results_file.write(str(step_forward))
```

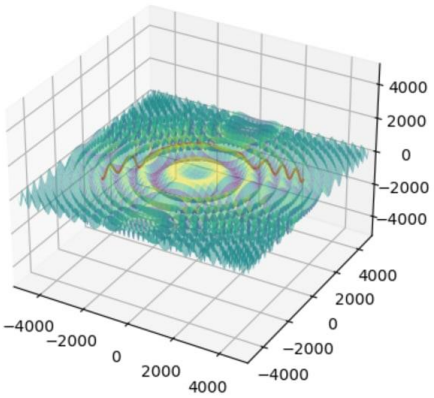
У програмі додатково реалізовані функції для визначення мінімальної відстані, пошуку суміжних точок і визначення відстані:

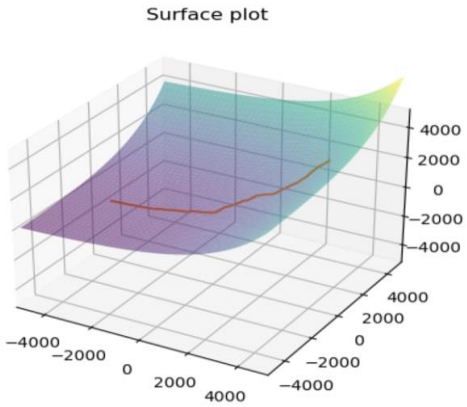
```python
def compute_distance(height1, height2, step):
    '''
    Returns a distance between points
using their heights and given step.

    >>> round(compute_distance(50, 60, 2))
    10
    >>> round(compute_distance(100, 200, 5))
    100
    '''

    #Finding distance between points using heights and step.
    distance = sqrt((height1 - height2)**2 + step**2)
    return distance

def get_minimum(distance1, distance2, ancestors, element, end):
    '''
    Returns a minimum distance.

    >>> ancestors = [[(2,4), (2,5), (3,7), (4,5)]]
    >>> get_minimum(50, 60, ancestors, (2,2), (0,2))
    60
    >>> print(ancestors)
    [[(2, 4), (2, 5), (2, 2), (4, 5)]]
    >>> ancestors = [[(2,4), (2,5), (3,7), (4,5)]]
    >>> get_minimum(100, 60, ancestors, (2,2), (0,2))
    100
    >>> print(ancestors)
    [[(2, 4), (2, 5), (3, 7), (4, 5)]]
    '''

    #Compares data of distances and returns minimal existing.
    if distance1 is None:
        return distance2
    if distance1 > distance2:
        return distance1
    ancestors[end[0]][end[1]] = element
    return distance2
```

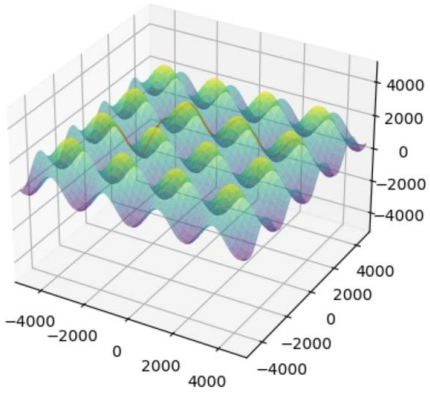
```python
def find_ribs(point, size):
    '''
    Returns a list
of the ends of the ribs formed
with point as a start

    >>> list(sorted(list(find_ribs((2,4), (10,10)))))
    [(1, 4), (2, 3), (2, 5), (3, 4)]
    >>> list(sorted(list(find_ribs((2,4), (2,5)))))
    [(1, 4), (2, 3), (3, 4)]
    '''

    #Finding sumizhni rebra of given point.
    ribs = set()
    index_x = point[0]
    index_y = point[1]

    #Finding rebra if the point is on the edge.
    if index_x == size[0] - 1:
        if index_y == 0:
            ribs.add((index_x - 1, index_y))
            ribs.add((index_x, index_y + 1))
        elif index_y == size[1] - 1:
            ribs.add((index_x - 1, index_y))
            ribs.add((index_x, index_y - 1))
        else:
            ribs.add((index_x - 1, index_y))
            ribs.add((index_x, index_y + 1))
            ribs.add((index_x, index_y - 1))
        return ribs
```

```python
    if index_x == 0:
        if index_y == 0:
            ribs.add((index_x + 1, index_y))
            ribs.add((index_x, index_y + 1))
        elif index_y == size[1] - 1:
            ribs.add((index_x + 1, index_y))
            ribs.add((index_x, index_y - 1))
        else:
            ribs.add((index_x + 1, index_y))
            ribs.add((index_x, index_y + 1))
            ribs.add((index_x, index_y - 1))
        return ribs

    #Finding rebra if the point is inside the grid.
    if index_y == 0:
        ribs.add((index_x + 1, index_y))
        ribs.add((index_x - 1, index_y))
        ribs.add((index_x, index_y + 1))
    elif index_y == size[1] - 1:
        ribs.add((index_x + 1, index_y))
        ribs.add((index_x - 1, index_y))
        ribs.add((index_x, index_y - 1))
    else:
        ribs.add((index_x + 1, index_y))
        ribs.add((index_x - 1, index_y))
        ribs.add((index_x, index_y - 1))
        ribs.add((index_x, index_y + 1))
    return ribs
```

Результати тестів:








```
Time: 50.11477589607239

Time: 57.81396532058716

Time: 55.306593894958496
```

Виконали:

Проектування і код:

1)Броницький Михайло
2)Роман Кипибіда
3)Трескот Вадим
4)Ростик Сидор
5)Гоєв Олексій

Репозиторій:

Гоєв Олексій

Звіт і тестування:

Роман Кипибіда

Feedback: kcus stsetcod