УКРАЇНСЬКИЙ КАТОЛИЦЬКИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНИХ НАУК

Пошук найкоротшого шляху між двома точками поверхні

Автори:
Притула Матвій
Снятинська Олена
Франко Микола
Серивко Остап
Бурак Василь

21 грудня 2021

1. Вступ.

У цьому проекті реалізовано алгоритм пошуку найкоротшого шляху між двома точками поверхні.

Алгоритми пошуку шляху важливі, оскільки вони використовуються в таких програмах, як карти Google, системи супутникової навігації, маршрутизація пакетів через Інтернет. Використання алгоритмів пошуку шляху не обмежується лише навігаційними системами. Загальну ідею можна застосувати і до інших додатків, зокрема у комп'ютерних іграх. У іграх, як і у житті, ми часто хочемо знайти шляхи з одного місця в інше, причому найкоротшу відстань. В таких випадках було б доцільно врахувати час у дорозі. Є кілька алгоритмів для знаходження найкоротшої відстані між двома точками на поверхні. Найпопулярнішими з них є алгоритми Дейкстри та алгоритм A*. Алгоритм A* покращує алгоритм найкоротшого шляху Дейкстри, включаючи додаткову інформацію за допомогою евристичної функції, яка визначає, які шляхи слід досліджувати далі. Завдяки цій оптимізації найкоротші шляхи знаходять швидше. Алгоритм A* можна використовувати для пошуку найкоротших шляхів між окремими парами місць, де відомі координати GPS.

Алгоритм A* реалізовано на мові Python. Використовуючи методи функціональної декомпозиції, алгоритм розбито на менші підзавдання.

2. Псевдокод.

```
Initialize 2 additional double lists to store [g,h] and previous
vertex
Count g,h,f value for start_vertex
Add g,h,f values to the storing double list at specific place
# starting vertex does not have previous (parent) :(
Initialize close list  # to keep track of the vertices we have
already visited or haven't yet
Add starting vertex to the open list
While open list is not empty:
   • set curr_vertex via the smallest if value in open list
   • count g,h,f values for curr_vertex
   • add curr_vertex to the close list  # don't need to check it
     anymore
   • if curr_vertex is final destination return path
   • find vertices adjacent to the curr_vertex
```

```
        for child in adjacent_list:
            If child is not in closed list:
                count g_value of the child
                if g_value of the child is higher than its previous
                g_value
                refresh child's g,h,f values
                set child's parent as the curr_vertex
                add child to the open list
If the path does not exist return None
```

Алгоритм A* описується однією простою формулою: F = G + H.

G – величина, що є ціною через всіх пройдених ребер починаючи з початкової вершини закінчуючи поточною.

H – величина, що є оціночною відстанню від поточної вершини до кінцевої. Обчислюється за формулою manhattan distance.

Для поточної вершини вибираються суміжні і додаються в множину відкритих вершин. Для кожної з вершин обчислюється G, H і, зокрема їхня сума, F. Поточна вершина додається в множину закритих вершин, щоб її не перевіряти ще раз і не повертатись до неї.

Після цього вибирається найкоротший шлях (F distance) і переходить на вершину з найменшим F. Проте, інші вершини не закриваються. Вони закриються тоді коли алгоритм пройде по них і побачить, що їхня F величина не підходить. Таким чином, алгоритм просувається вперед до моменту поки не дійде до фінальної вершини.

Перевагою алгоритму A* над алгоритмом Дейкстри є те, що A* знаходить лише найкоротший шлях, а не знаходить всі варіації шляху.

Довжиною найкоротшого шляху буде мінімальне G distance в вершині кінцевого призначення.

Методом функціональної декомпозиції, алгоритм був поділений на менші підзавдання:

- Розрахунок дистанції H (calc_heuristic())

```python
def calc_heuristic(curr_vertex, finish_vertex, graph, step):
    """
    calculate heuristic distance for the current vertex
    PS: that's manhattan distance which counts based on x,y,z
    coordinates of the vertex
    :param curr_vertex:
    :param finish_vertex:
    :return:
    """

    x_difference = abs(curr_vertex[0] - finish_vertex[0])
    y_difference = abs(curr_vertex[1] - finish_vertex[1])
    z_start = graph[curr_vertex[1]][curr_vertex[0]]
    z_finish = graph[finish_vertex[1]][finish_vertex[0]]
    z_difference = abs(z_start - z_finish)
    return (x_difference + y_difference) * step + z_difference
```

- Знаходження відстані F (calc_f_value())

```python
def calc_f_value(g_distance, heuristic_distance):
    """
    literally the sum of both arguments
    :param g_distance:
    :param heuristic_distance:
    :return: the sum of two given arguments
    """
    return g_distance + heuristic_distance
```

- Знаходження сусідніх вершин до поточної (find_adjastent())

```python
def find_adjacent(curr_vertex, graph):
    """
    this function find ALL the vertexes that are connected to the current
    one based on the next condition:
    only those vertexes are connected that have a difference=1 of only one parameter (either x or y)
    :param graph:
    :param curr_vertex:
    :return: adjacent_list
    adjacent_list is gonna be a list of tuples
    each tuple consists of to integers: x and y index of a vertex
    ATTENTION:
    Please take into account the fact that not all vertexes are going to have
    four connected vertexes with them as some of them are going to be the ones on the side
    meaning they have only three adjacent vertexes
    and some of them are going to be in the corner
    meaning they only have two adjacent vertexes
    """
    adjacent_list = set()
    x, y = curr_vertex[0], curr_vertex[1]
    if x - 1 >= 0:
        adjacent_list.add((x - 1, y))
    if y - 1 >= 0:
        adjacent_list.add((x, y - 1))
    if x + 1 < len(graph):
        adjacent_list.add((x + 1, y))
    if y + 1 < len(graph[0]):
        adjacent_list.add((x, y + 1))
    return adjacent_list
```

- Повернення побудованого шляху (get_path())

```python
def get_path(current_vertex, prev_vertex):
    """
    this function returns the path from the nested list of parents nodes
    :param current_vertex:
    :param prev_vertex:
    :return: path: list of tuples
    """
    path = []
    node = current_vertex
    while node is not None:
        path.append(node)
        node = prev_vertex[node[-1]][node[0]]
    return path[::-1]  # reversed path as we store it backwards
```

- Функція зберігання даних та їх обробки (path_finding())

```python
def path_finding(graph, start_vertex, finish_vertex, step):
    """
    that is the function which ties up all the other functions
    and goes before main
    there we initialise the dictionary of our open vertexes
    the key will be the tuple of two elements(x,y) of a vertex
    and the value is going to be a list
    where:
    the first element: g_distance of a vertex
    the second element: heuristic distance of a vertex
    the third element: f_value of a vertex
    the fourth element: tuple of the previous vertex
    PS: previous vertex means the vertex which we got to the current one from
    :param graph:
    :param step:
    :param start_vertex:
    :param finish_vertex:
    :return:
    """
    height, width = len(graph), len(graph[0])

    prev_vertexes = [[None for _ in range(width)] for _ in range(height)]  # to get the path later
    g_and_h = [[[float("inf"), float("inf")] for _ in range(width)] for _ in range(height)]  # to store G and H

    start_g = 0
    start_h = calc_heuristic(start_vertex, finish_vertex, graph, step)
    g_and_h[start_vertex[-1]][start_vertex[0]] = [start_g, start_h]
    start_f = start_h + start_g
    open_heap = []
    closed_set = set()
    heapq.heappush(open_heap, [start_f, start_vertex])
    while open_heap:
        curr_vertex = heapq.heappop(open_heap)[-1]  # set the current vertex via smallest f
        curr_g = g_and_h[curr_vertex[-1]][curr_vertex[0]][0]
        closed_set.add(curr_vertex)
```

```python
        # if got to the finish
        if curr_vertex == finish_vertex:
            print(f"Length: {curr_g}")
            return get_path(curr_vertex, prev_vertexes)

        # look through all the adjacent vertexes
        children = find_adjacent(curr_vertex, graph)
        for child in children:
            if child not in closed_set:
                z_dif = graph[curr_vertex[-1]][curr_vertex[0]] - graph[child[-1]][child[0]]
                child_g = curr_g + math.hypot(step, z_dif)
                if child_g < g_and_h[child[-1]][child[0]][0]:
                    child_h = calc_heuristic(child, finish_vertex, graph, step)
                    child_f = child_g + child_h
                    g_and_h[child[-1]][child[0]] = [child_g, child_h]
                    prev_vertexes[child[-1]][child[0]] = (curr_vertex[0], curr_vertex[-1])
                    heapq.heappush(open_heap, (child_f, child))
    print("The path does not exist")
    return None
```

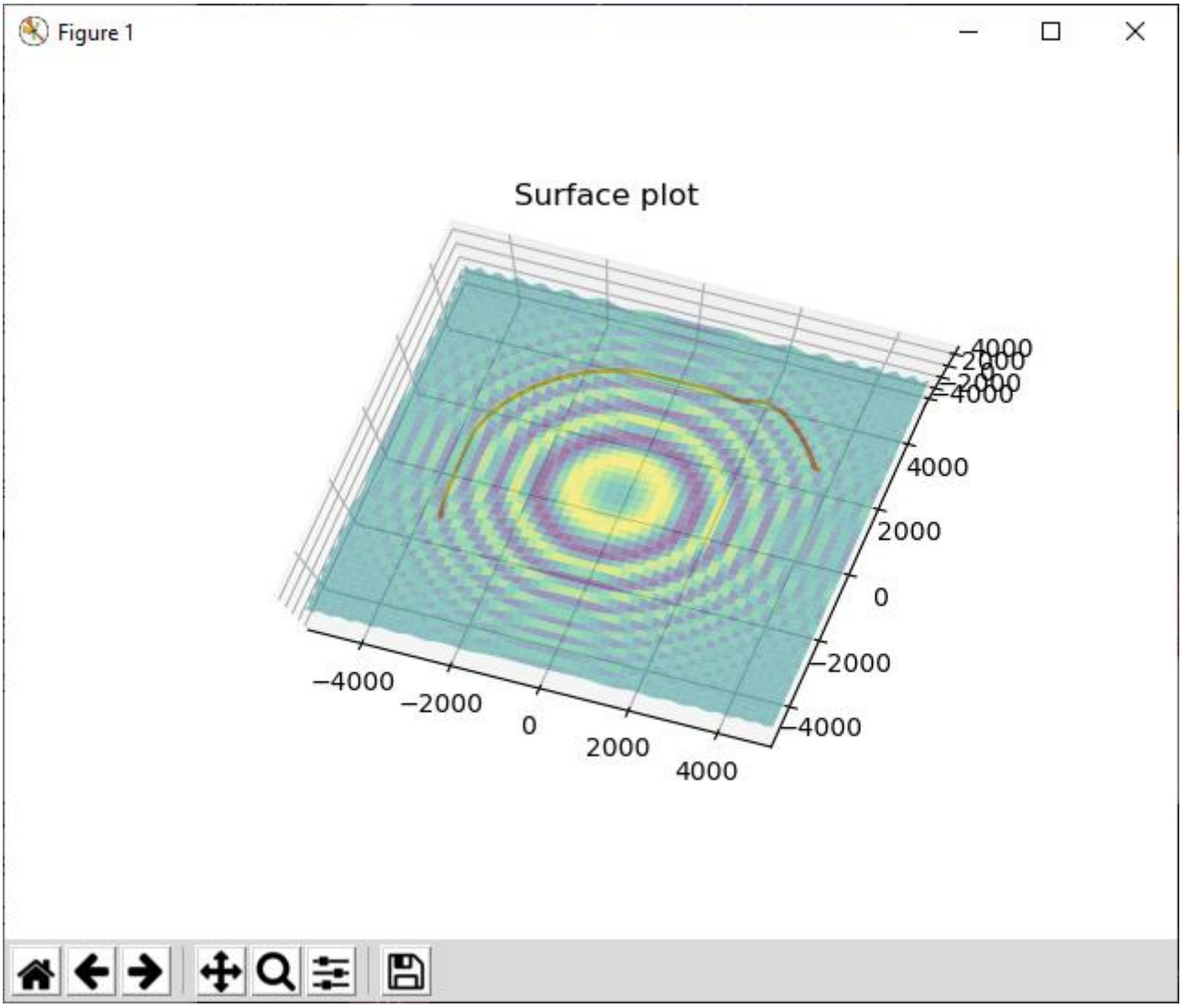
- Функції отримання і збереження даних (read_csv() & main())

```python
def read_csv(path_to_file):
    """
    the function which turns csv file into the nested list
    :param path_to_file:
    :return: starting node, ending node, the graph as a list of lists
    """
    graph = []
    with open(path_to_file, "r", encoding="utf-8") as my_file:
        csv_reader = csv.reader(my_file, delimiter=" ")
        for idx, row in enumerate(csv_reader):
            if idx > 2:
                for k in range(len(row)):
                    row[k] = float(row[k])
                graph.append(row)
            elif idx == 2:
                ending_vertex = (int(row[-2]), int(row[-1]))
            elif idx == 1:
                starting_vertex = (int(row[-2]), int(row[-1]))
            elif idx == 0:
                step = int(row[-1])
    return step, starting_vertex, ending_vertex, graph
```

```python
def main():
    """
    the function connects all the code and writes the result into specified txt file
    """
    start_time = time.time()
    info = read_csv("examples/example1.csv")
    graph = info[-1]
    starting_node = info[1]
    ending_node = info[2]
    step = info[0]
    the_path = path_finding(graph, starting_node, ending_node, step)

    with open("results.txt", "w", encoding="utf-8") as res_file:
        res_file.write(str(the_path))

    finish_time = time.time()
    print(f"Time taken: {round(finish_time - start_time)} sec")
```



3. Висновки.

Алгоритм пошуку найкоротшої відстані широко застосовується у різних аспектах технологічного життя. Наприклад, NPC-герої ігор найчастіше пересуваються саме таким чином.

A* не відвідує кожну вершину графу, тому він є більш оптимізований, ніж, наприклад алгоритм Дейкстра.

Цей алгоритм вибирає вершину, з найбільшим потенціалом, яка може привести нас у фінальну точку.

Що точніша оціночна відстань - то швидше знаходиться найкоротший шлях

Масив відкритих вершин зберігається, як пріоритетна черга (priority queue)