Sprawozdanie z zadania 10

Adrian Wrona 179993

5 December 2024

Spis treści

Treść zadania

Dla punktów płaszczyzny, których współrzędne x i y są przechowywane w dwóch tablicach o długości n utworzyć tablicę, która pod indeksem *i*- tym indeksem będzie przechowywać indeks najbliższego sąsiada *i*- tego punktu.

Przykład:

Wejście

[0, 1, -2, -1, 10]

[0, 2, -3, -10, 9]

Wyjście

[1, 0, 0, 2, 1]

1 Rozwiązanie problemu za pomocą algorytmu typu brute force

1.1 Analiza problemu

W podejściu (Zgodnie z treścią zadania) stworzymy tablicę dla współrzędnych x i y oraz tablicę, która będzie przechowywać pozycję najbliższego sąsiada. Aby znaleźć właściwe współrzędne pozycji skorzystamy z funkcji obliczającej odległość euklidesową (odległość pitagorejską) między punktami (x1, y1) i (x2, y2). Zawrzemy ją w kodzie programu, aby porównywać odległości kolejnych punktów w tabeli.

Postać wzoru, na którym bazujemy tę funkcję :

$$\sqrt{(x_1 - y_1)^2 + (x_2 + y_2)^2}$$

W kolejnych pętlach porównujemy następne elementy, aż nie znajdziemy najbliższych sąsiadów, dla każdego z elementów tablicy. Po dokonaniu porównań zapiszemy pozycję najbliższego sąsiada każdego elementu w zmiennej "nearest index". Na końcu wypiszemy wszystkie potrzebne wartości. Podczas porównania nie porównujemy danego punktu z samym sobą.

Zakładamy, że ilość punktów nie będzie większa np. od 100, aby ograniczyć działanie programu do danych wartości i podać rozmiary tablicy x i y. W przypadku, gdy jest więcej niż jeden punkt w równej odległości od trzeciego punktu program podaje pierwszy znaleziony punkt.

1. Dane Wejściowe:

   Danymi wejściowymi naszego algorytmu uczynimy:

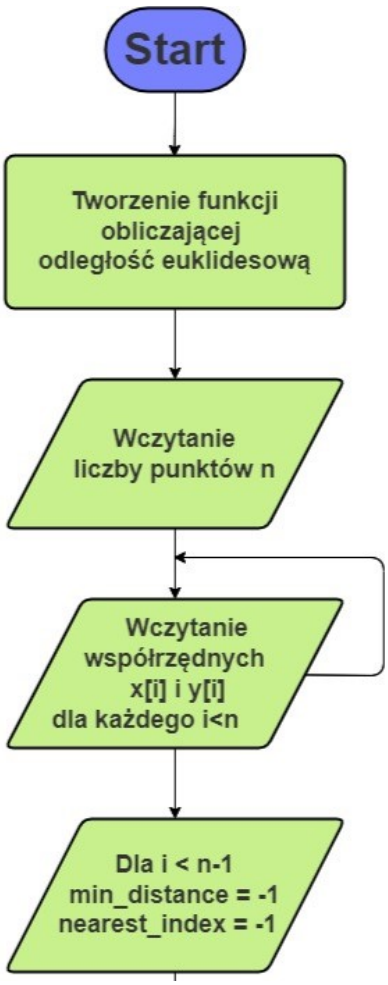
   - Ilość punktów w strukturach danych typu tablica (x i y) , czyli zmienną n.
   - Wartości kolejnych elementów ciągu w tablicy x i y o ich ilości zależnej od zmiennej n.

2. Dane Wyjściowe:

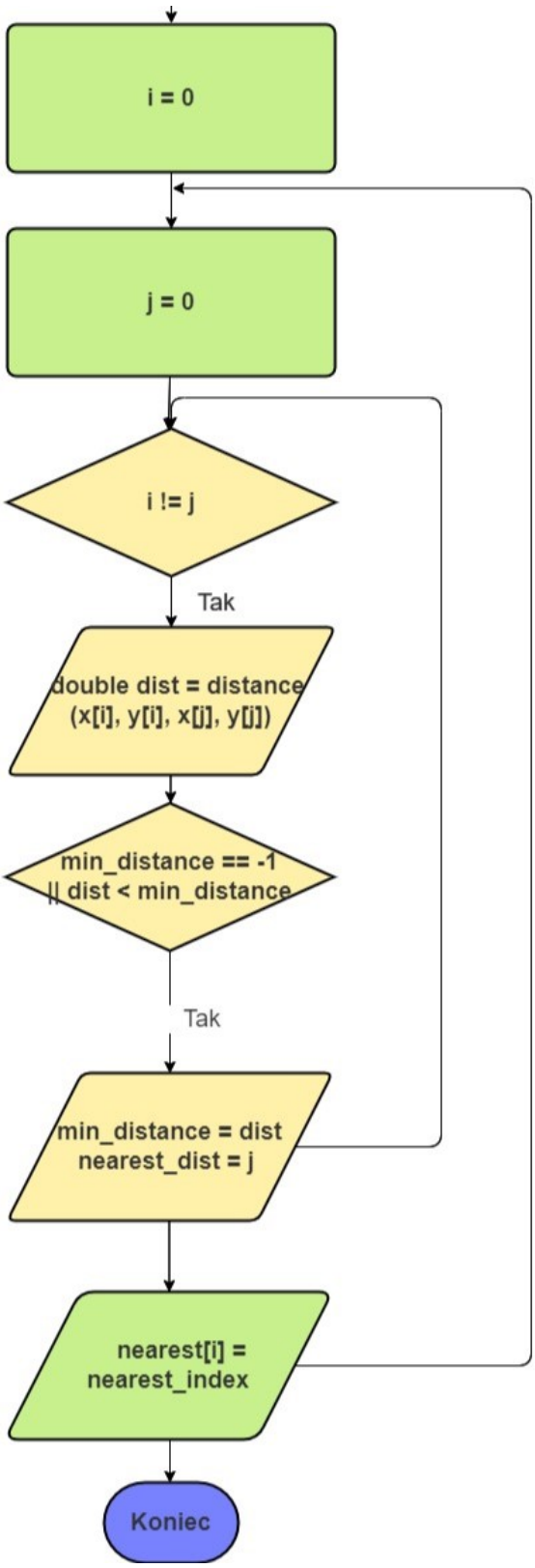
   Algorytm zwróci dane wartości za pomocą zmiennej nearest[i].

1.2 Schemat blokowy algorytmu

Algorytm zapisany w postaci schematu blokowego mógłby przedstawiać się następująco:



Rysunek 1: Schemt blokowy algorytmu cz. 1



Rysunek 2: Schemat blokowy algorytmu cz. 2

Zauważmy, że do algorytmu dodaliśmy dodatkowe zmienne, które pozwolą nam śledzić najbliższego sąsiada i jego pozycję (zmienne dist, min distance, nearest dist oraz nearest index) i wskaźniki(i oraz j), które pomogą nam poruszanie się po tablicach.

```
 Wejscie:
         n         // liczba punktow
         x[i]      //wspolrzedne punktu w tabeli x
         y[i]      //wspolrzedne punktu w tabeli y

 Wyjscie:
         nearest[i]  // indeks najblizszego sasiada

     Wykorzystujemy wzor na odleglosc euklidesowa jako podstawa
kodu

     for i=0 i<n i++
     {
      min_distance= -1
      nearest_index= -1
         for j=0 j<n j++
         if( i!=j)
             dist=d(x,y)
         if(min_distance == -1 || dist<min_distance)
             min_distance=dist
             nearest_index=j

     nearest[i]=nearest_index
     }
 Wypisz nearest[i] dla wszystkich wspolrzednych
```

Pseudokod w zapisie zawiera dwie petle for (gdzie mniejsza znajduje się w wiekszej)

1.3 "Ołówkowe" sprawdzenie algorytmu

Dane Wejściowe:

x [0, 1, -2, -1, 10]

y [0, 2, -3, -10, 9]

Współrzędne punktów to:

1. (0, 0)

2. (1, 2)

3. (-2, -3)

4. (-1, -10)

5. (10, 9)

Obliczamy odległości dla tych współrzędnych, sprawdzamy czy dana odległość jest mniejsza od poprzedniej wartości i powtarzamy to za nim znajdziemy najmniejszą odległość:

1. z 2.	$\sqrt{5}$	2. z 1.	$\sqrt{5}$	3 z 1.	$\sqrt{12}$	4. z 1	$\sqrt{101}$	5. z 1.	$\sqrt{181}$
1. z 3.	$\sqrt{12}$	2. z 3.	$\sqrt{34}$	3 z 2.	$\sqrt{34}$	4. z 2.	$2\sqrt{37}$	5. z 2.	$\sqrt{130}$
1. z 4.	$\sqrt{101}$	2. z 4.	$2\sqrt{37}$	3 z 4.	$5\sqrt{2}$	4. z 3.	$5\sqrt{2}$	5. z 3.	$12\sqrt{2}$
1. z 5.	$\sqrt{181}$	2. z 5.	$\sqrt{130}$	3 z 5.	$12\sqrt{2}$	4. z 5.	$\sqrt{482}$	5. z 4.	$\sqrt{482}$

Punkt	1.	2.	3.	4.	5.
Najbliższy sąsiad	2.	1.	1.	3.	2.

1.4 Teoretyczne oszacowanie złożoności algorytmu

Podstawą programu jest porównywanie wartości i i j za pomocą iteracji zewnętrznej i wewnętrznej , analizując działanie programu możemy zauważyć że złożoność czasowa algorytmu wynosi:

$$O(n^2)$$

Obecny algorytm będzie dość sprawnie wyszukiwał najbliższego sąsiada, jednak przy dużej ilości zmiennych czas działania programu znacznie się wydłuży z powodu niemożliwości uniknięcia sprawdzenia każdego elementu po kolei. Możemy zmodyfikować program, aby wyszukiwał to szybciej. Aby zmniejszyć czas wyszukiwania i doprowadzić złożoność czasową do postaci logarytmicznej można wykorzystać algorytm oparty na algorytmie dziel

i rządź, jednakże mimo prób nie udało mi się stworzyć algorytmu, który zadowalająco realizował to zadanie (prawdopodobnie z powodu braku wystarczającej wiedzy lub umiejętności)

Z tego powodu ostateczna postać kodu będzie bazować na algorytmie wyszukiwania z podpunktu 1.2.

2 Praca nad kodem

2.1 Prosta implementacja wymyślonego algorytmu w wybranym środowisku i języku oraz eksperymentalne potwierdzenie wydajności (złożoności obliczeniowej) algorytmu.

Cały kod umieszczamy w jednym pliku, a potem wywołujemy program i prezentujemy działanie programu na podstawie treści zadania.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#define MAX_POINTS 100000  // Maksymalna liczba punktow

// Funkcja obliczajaca odleglosc euklidesowa
// (odleglosc pitagorejska) miedzy dwoma punktami (x1, y1) i (x2, y2)

double distance(double x1, double y1, double x2, double y2) {
    return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

int main() {

    srand(time(NULL));

    int n=100000;  // Liczba punktow
    //printf("Podaj liczbe punktow: ");
   //scanf("%d", &n);
   // printf("\n");
  double x[MAX_POINTS], y[MAX_POINTS];  // Tablice przechowujace
    wspolrzedne punktow
    int nearest[MAX_POINTS];  // Tablica przechowujaca pozycje
    najblizszych sasiadow

    // Wczytanie wspolrzednych punktow
    for (int i = 0; i < n; i++) {
      x[i] = rand() % 100-50;
      y[i] = rand() % 100-50;
      //printf("Punkt %d: x= %.f, y= %.f \n",i+1, x[i], y[i]);
```

```
29
30
31      }
32      // Dla kazdego punktu znajdz najblizszego sasiada
33      for (int i = 0; i < n; i++) {
34          double min_distance = -1;  // Minimalna odleglosc
35          // (zaczynamy od wartosci -1, aby znalesc pierwsza odleglosc)
36          int nearest_index = -1;  // Indeks najblizszego punktu
37
38          for (int j = 0; j < n; j++) {
39              if (i != j) {  // Nie porownujemy punktu z samym soba
40                  double dist = distance(x[i], y[i], x[j], y[j]);
41
42                  // Sprawdzamy, czy znalezlismy mniejsza odleglosc
43                  if (min_distance == -1 || dist < min_distance) {
44                      min_distance = dist;
45                      nearest_index = j;
46                  }
47              }
48          }
49
50          // Zapisujemy indeks najblizszego punktu
51          nearest[i] = nearest_index;
52      }
53
54      // Wyswietlanie wynikow
55      printf("\nIndeksy najblizszych sasiadow:\n");
56      for (int i = 0; i < n; i++) {
57          printf("Dla punktu %d najblizszy sasiad to punkt %d \n", i +
    1, nearest[i] + 1);
58          // dodajemy 1 do najblizszego sasiada by zgadzala sie z
    iloscia punktow
59      }
60
61      return 0;}
```

```
1
2 Kod Programu:
3
4 Podaj liczbe punktow: 5
```

```
Punkt 1:
Podaj wspolrzedne punktu(x): 0
Podaj wspolrzedne punktu(y): 0

Punkt 2:
Podaj wspolrzedne punktu(x): 1
Podaj wspolrzedne punktu(y): 2

Punkt 3:
Podaj wspolrzedne punktu(x): -2
Podaj wspolrzedne punktu(y): -3

Punkt 4:
Podaj wspolrzedne punktu(x): -1
Podaj wspolrzedne punktu(y): -10

Punkt 5:
Podaj wspolrzedne punktu(x): 10
Podaj wspolrzedne punktu(y): 9
```

2.1.1 Testy

Będziemy sprawdzać wydajność kodu dla wartości

n= 2500, 5000, 10000,20000, 300000, 40000, 50000, 60000, 70000, 80000

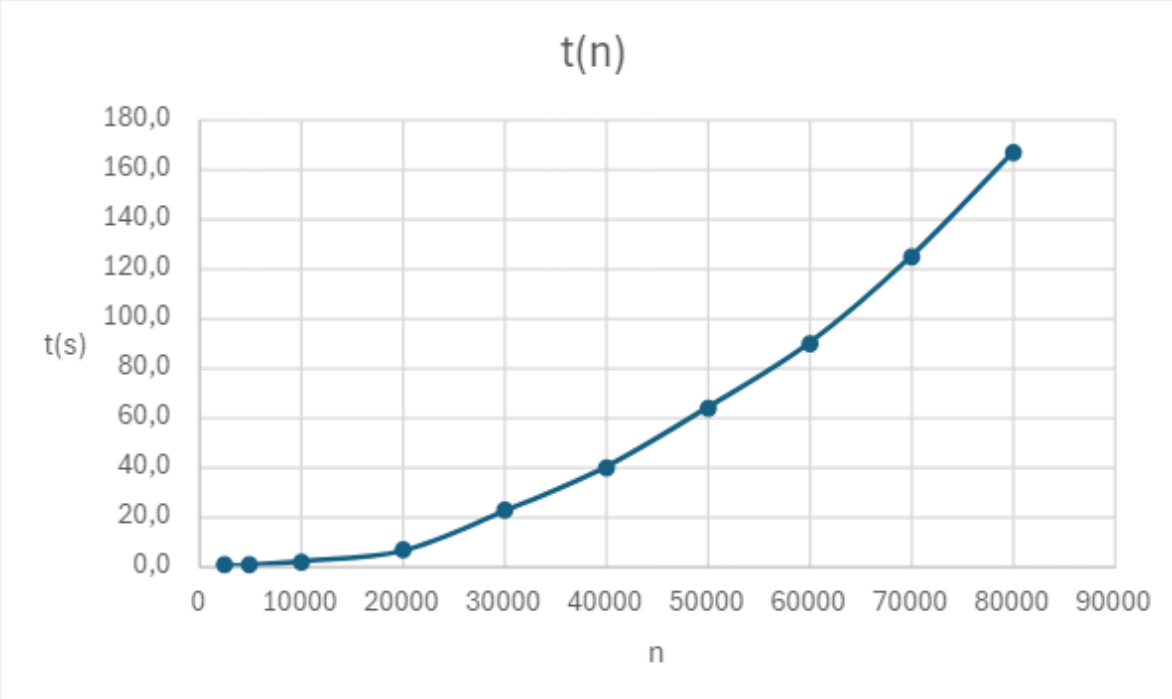
W tym celu dodamy funkcje, które będą generować liczby losowe od np. [0 do 100]. Zmodyfikujemy również kod, aby wyświetlał czas jaki potrzebuje program na wyszukanie wszystkich sąsiadów dla danej ilości punktów.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Funkcja obliczajaca odleglosc euklidesowa miedzy dwoma punktami
double e_distance(double x1, double y1, double x2, double y2) {
    return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}

```

```
// Funkcja do obliczenia najblizszego sasiada
void find_nearest_neighbors(int n, double x[], double y[], int
    neighbors[]) {
     for (int i = 0; i < n; i++) {
         double min_distance = 1e10;  // Bardzo duza wartosc poczatkowa
         int nearest = -1;

         // Sprawdzamy odleglosc kazdego punktu od innych punktow
         for (int j = 0; j < n; j++) {
             if (i != j) {  // Nie porownujemy punktu z samym soba
                 double distance = e_distance(x[i], y[i], x[j], y[j]);
                 if (distance < min_distance) {
                     min_distance = distance;
                     nearest = j;
                 }
             }
         }
         neighbors[i] = nearest;  // Zapisujemy indeks najblizszego
    sasiada
     }
}

int main() {
 int n[] = { 2500, 5000, 10000, 20000, 30000, 40000, 50000, 60000,
    70000, 80000};
 // Liczba punktow do testow
 srand(time(NULL));  // Inicjalizacja generatora liczb losowych
 for (int t = 0; t < 10; t++) {
 int points_count = n[t];  // Liczba punktow w biezacym tescie

  double x[points_count], y[points_count];
 int neighbors[points_count];

 // Generowanie losowych wspolrzednych punktow
 printf("\nGenerowanie wspolrzednych punktow dla n = %d...\n",
    points_count);
 for (int i = 0; i < points_count; i++) {
  // Losowanie wspolrzednych x i y w zakresie [0, 100]
 x[i] = rand() % 100 ;  // Liczby losowe w przedziale [0, 100]
 y[i] = rand() % 100 ;  // Liczby losowe w przedziale [0, 100]
```

```
48      }
49
50          // Zmierzenie czasu wykonania
51          clock_t start = clock();
52
53          // Wywolanie funkcji do znalezienia najblizszych sasiadow
54          find_nearest_neighbors(points_count, x, y, neighbors);
55
56          clock_t end = clock();
57          double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
58
59          printf("Czas wykonania dla %d punktow: %f sekundy\n",
    points_count, time_taken);
60
61
62      }
63
64      return 0;
65  }
```

Na podstawie wprowadzonych danych (n) i czasu działania programu (t(s)) możemy stworzyć wykres funkcji, który ma kształt paraboli o prawdopodobnej funkcji: $t(n) = 2.298 * 10^{(}-8) * n$.



Rysunek 3: Wykres oparty na funkcji $t(n) = 2.298 * 10^{(} - 8) * n$

Podsumowanie

Program mimo bycia mniej optymalnym przy większej ilości danych, spełnia swoją rolę i właściwie odnajduje najbliższego sąsiada danego punktu.