

**WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI**
POLITECHNIKI RZESZOWSKIEJ

Sprawozdanie z zadania 10

Adrian Wrona 179993

11 December 2024

Spis treści

Treść zadania	3
1. Rozwiązanie problemu za pomocą algorytmu typu brute force	4
1.1. Analiza problemu	4
1.2. Schemat blokowy algorytmu	5
1.3. Zapis algorytmu w pseudokodzie	6
1.4. "Ołówkowe" sprawdzenie algorytmu	7
1.5. Teoretyczne oszacowanie złożoności algorytmu	7
Ponowna analiza problemu	7
2. Praca nad kodem	8
2.1. Prosta implementacja wymyślnego algorytmu w wybranym środowisku i języku oraz eksperymentalne potwierdzenie wydajności (złożoności oblicze- niowej) algorytmu.	8
2.1.1. Testy	10
Podsumowanie	13

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:

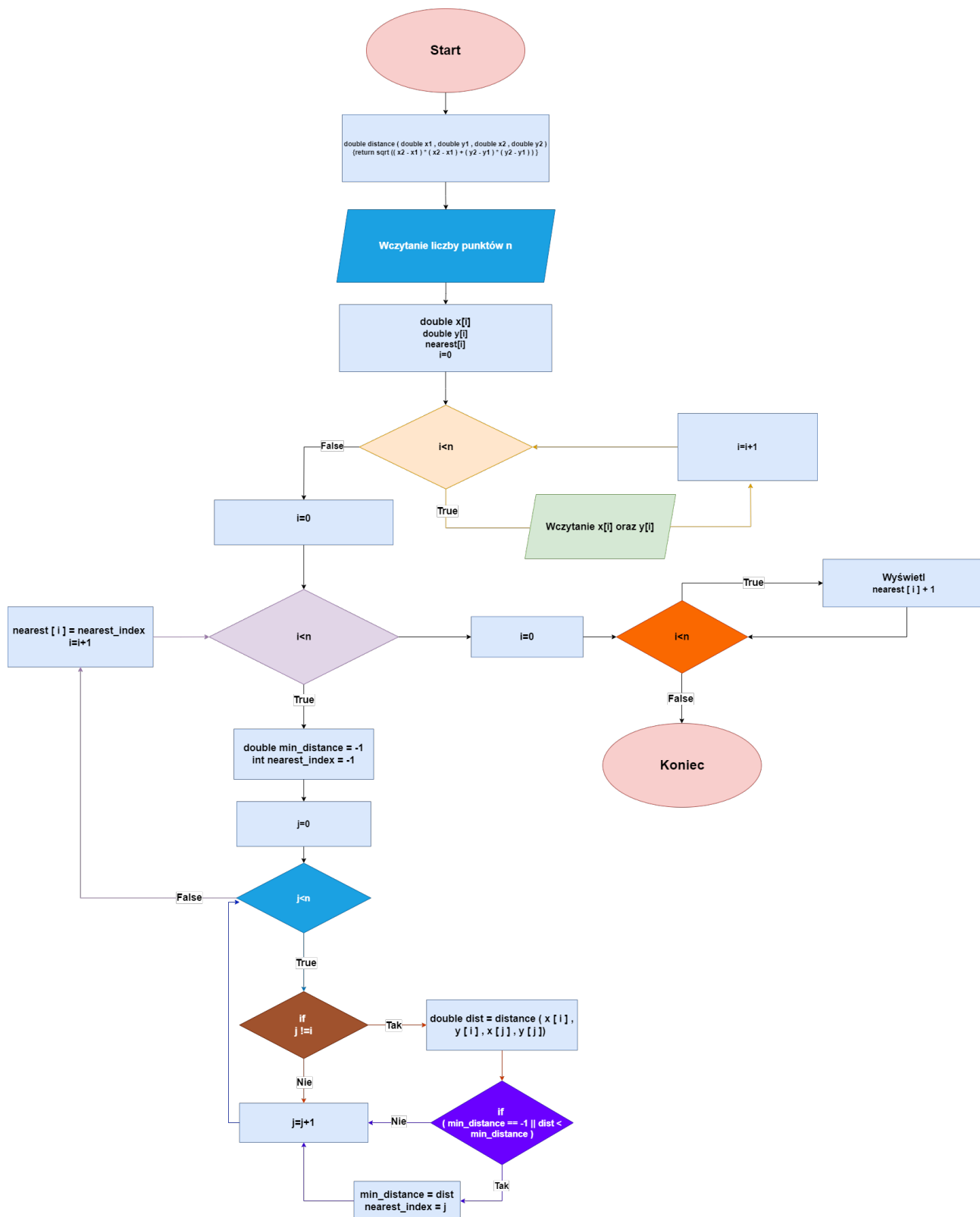
- Ilość punktów w strukturach danych typu tablica (x i y) , czyli zmienną n.
- Wartości kolejnych elementów ciągu w tablicy x i y o ich ilości zależnej od zmiennej n.

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

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.

1.3 Zapis algorytmu w pseudokodzie

```
1   Wejscie:
2       n          // liczba punktow
3       x[i]       //wspolrzedne punktu w tabeli x
4       y[i]       //wspolrzedne punktu w tabeli y
5
6   Wyjscie:
7       nearest[i] // indeks najblizszego sasiada
8
9       Wykorzystujemy wzor na odleglosc euklidesowa jako podstawa
10      kodu
11
12      for i=0 i<n i++
13      {
14          min_distance= -1
15          nearest_index= -1
16          for j=0 j<n j++
17              if( i!=j)
18                  dist=d(x,y)
19                  if(min_distance == -1 || dist<min_distance)
20                      min_distance=dist
21                      nearest_index=j
22
23          nearest[i]=nearest_index
24      }
25
26      Wypisz nearest[i] dla wszystkich wspolrzednych
```

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

1.4 "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:

1.5 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 niemożności 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 wymyślnego 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.

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #define MAX_POINTS 100000 // Maksymalna liczba punktów
5
6 // Funkcja obliczająca odległość euklidesowa
7 // (odległość pitagorejska) między dwoma punktami (x1, y1) i (x2, y2)
8 double distance(double x1, double y1, double x2, double y2) {
9     return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
10 }
11
12 int main() {
13
14     srand(time(NULL));
15
16     int n=100000; // Liczba punktów
17     //printf("Podaj liczbę punktów: ");
18     //scanf("%d", &n);
19     // printf("\n");
20     double x[MAX_POINTS], y[MAX_POINTS]; // Tablice przechowujące
        współrzędne punktów
21     int nearest[MAX_POINTS]; // Tablica przechowująca pozycje
        najbliższych sąsiadów
22
23     // Wczytanie współrzędnych punktów
24     for (int i = 0; i < n; i++) {
25         x[i] = rand() % 100-50;
26         y[i] = rand() % 100-50;
27         //printf("Punkt %d: x= %.f, y= %.f \n",i+1, x[i], y[i]);
28
```



```

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

```

```

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

```

```

6 Punkt 1:
7 Podaj wspolrzedne punktu(x): 0
8 Podaj wspolrzedne punktu(y): 0
9
10 Punkt 2:
11 Podaj wspolrzedne punktu(x): 1
12 Podaj wspolrzedne punktu(y): 2
13
14 Punkt 3:
15 Podaj wspolrzedne punktu(x): -2
16 Podaj wspolrzedne punktu(y): -3
17
18 Punkt 4:
19 Podaj wspolrzedne punktu(x): -1
20 Podaj wspolrzedne punktu(y): -10
21
22 Punkt 5:
23 Podaj wspolrzedne punktu(x): 10
24 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.

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5 #include <time.h>
6
7 // Funkcja obliczajaca odleglosc euklidesowa miedzy dwoma punktami
8 double e_distance(double x1, double y1, double x2, double y2) {
9     return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
10 }
11
12 // Funkcja do obliczenia najblizszego sasiada

```

```

13 void find_nearest_neighbors(int n, double x[], double y[], int
    neighbors[]) {
14     for (int i = 0; i < n; i++) {
15         double min_distance = 1e10; // Bardzo duza wartosc poczatkowa
16         int nearest = -1;
17
18         // Sprawdzamy odleglosc kazdego punktu od innych punktow
19         for (int j = 0; j < n; j++) {
20             if (i != j) { // Nie porownujemy punktu z samym soba
21                 double distance = e_distance(x[i], y[i], x[j], y[j]);
22                 if (distance < min_distance) {
23                     min_distance = distance;
24                     nearest = j;
25                 }
26             }
27         }
28         neighbors[i] = nearest; // Zapisujemy indeks najblizszego
        sasiada
29     }
30 }
31
32 int main() {
33     int n[] = { 2500, 5000, 10000, 20000, 30000, 40000, 50000, 60000,
        70000, 80000};
34     // Liczba punktow do testow
35     srand(time(NULL)); // Inicjalizacja generatora liczb losowych
36     for (int t = 0; t < 10; t++) {
37         int points_count = n[t]; // Liczba punktow w biezacym tesie
38
39         double x[points_count], y[points_count];
40         int neighbors[points_count];
41
42         // Generowanie losowych wspolrzednych punktow
43         printf("\nGenerowanie wspolrzednych punktow dla n = %d...\n",
            points_count);
44         for (int i = 0; i < points_count; i++) {
45             // Losowanie wspolrzednych x i y w zakresie [0, 100]
46             x[i] = rand() % 100 ; // Liczby losowe w przedziale [0, 100]
47             y[i] = rand() % 100 ; // Liczby losowe w przedziale [0, 100]
48         }

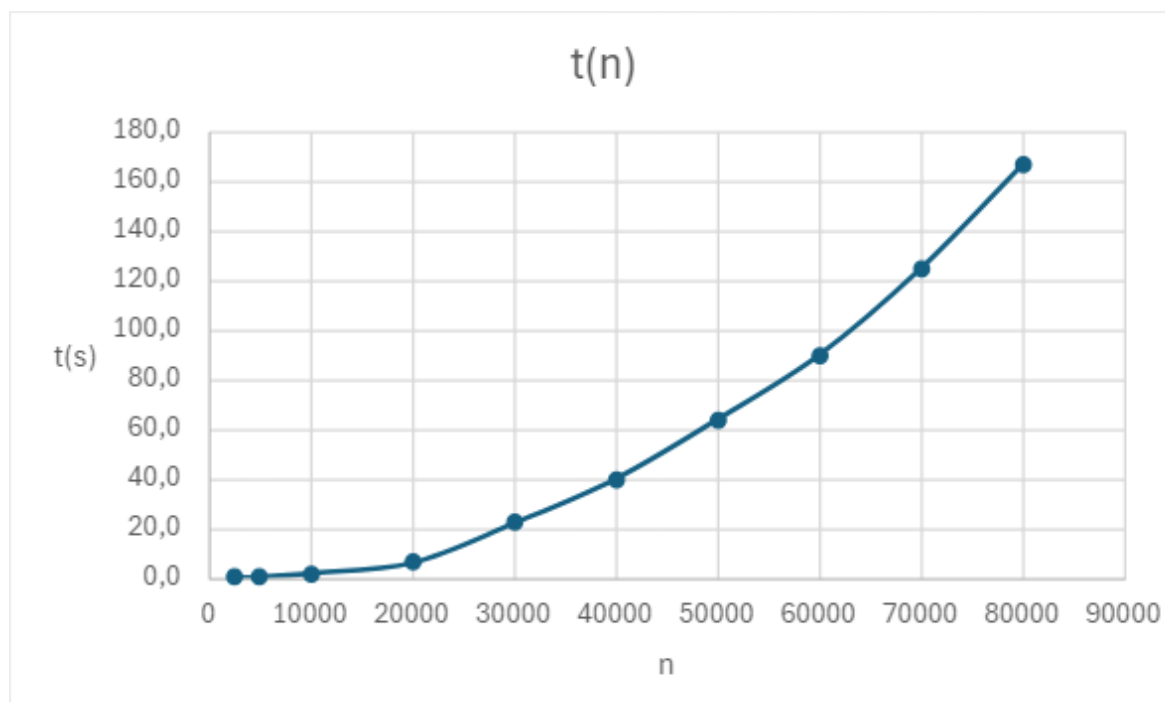
```

```

49
50     // Zmierzenie czasu wykonania
51     clock_t start = clock();
52
53     // Wywołanie funkcji do znalezienia najbliższych sąsiadów
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 punktów: %f sekundy\n",
60           points_count, time_taken);
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 2: Wykres oparty 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.