



**POLITECHNIKA
RZESZOWSKA**

im. IGNACEGO ŁUKASIEWICZA



**WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ**
POLITECHNIKI RZESZOWSKIEJ

Algorytmy i struktury danych

Autor: Jakub Szpila (179984) [Inżynieria i analiza danych]

Grupa: 7

Rzeszów, 26.01.2025 r.

Spis treści

1. Treść zadania	2
2. Etapy rozwiązywania problemu	3
2.1. Rozwiązanie - podejście pierwsze	3
2.1.1. Analiza problemu.....	3
2.1.1. Schemat blokowy algorytmu.....	4
2.1.3. Algorytm zapisany w pseudokodzie	5
2.1.4 Sprawdzenie poprawności algorytmu	6
2.1.5. Teoretyczne oszacowanie złożoności obliczeniowej	7
2.1.6 Implementacja algorytmu	8
3. Testy i wyniki eksperymentalne	10
3.1 Przykładowe dane wejściowe i wyjściowe	11
3.2 Testy wydajności	11
4. Wnioski.....	12

Streszczenie

Celem tej instrukcji jest przedstawienie poprawnego sposobu rozwiązywania zadania projektowego nr 21.

Niech dane będzie następujące zadanie:

1. Treść zadania

Dla tablicy $M \times N$ wypełnionej wartościami 0 i 1, znajdź liczbę znaków "plus" z jedynek ('plusem' jest krzyżyk z jedynek otoczony wyłącznie zerami).

Przykład

Wejście

```
[0 1 0 0 0 0 1 0 0 0 0 1]
[1 1 1 0 0 0 0 0 0 0 1 1]
[0 1 0 0 0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0]
[1 0 0 0 0 0 1 0 0 0 0 0]
[1 1 0 0 0 1 1 1 0 0 0 0]
[1 0 0 0 0 0 1 0 0 0 0 0]
```

Wyjście 2

Dane wejściowe znajdują się w pliku **dane.txt**, a wynikowy plik wyjściowy, zawierający liczbę poprawnych "plusów", zostanie zapisany w pliku **wynik.txt**.

2. Etapy rozwiązywania problemu

W trakcie rozwiązywania problemu najpierw skupiłem się na napisaniu kodu, odpowiadającemu poleceniu w otrzymanym zadaniu a następnie analizie czy można go wykonać wydajniej.

2.1. Rozwiązanie - podejście pierwsze

2.1.1. Analiza problemu

W zadaniu należy znaleźć liczbę znaków "plus" utworzonych z wartości 1 w dwuwymiarowej tablicy $M \times N$. Definicja "plusa" jest precyzyjna: w centrum musi znajdować się 1, a w każdej z czterech sąsiednich komórek (góra, dół, lewo, prawo) również muszą znajdować się 1. Otoczenie plusa (łącznie 8 komórek), zarówno w kierunku pionowym, jak i poziomym, musi składać się wyłącznie z zer.

Dane wejściowe:

- Tablica $M \times N$ zawierająca wartości 0 i 1.
- Wymiary i zawartość tablicy odczytane z pliku tekstowego.

Dane wyjściowe:

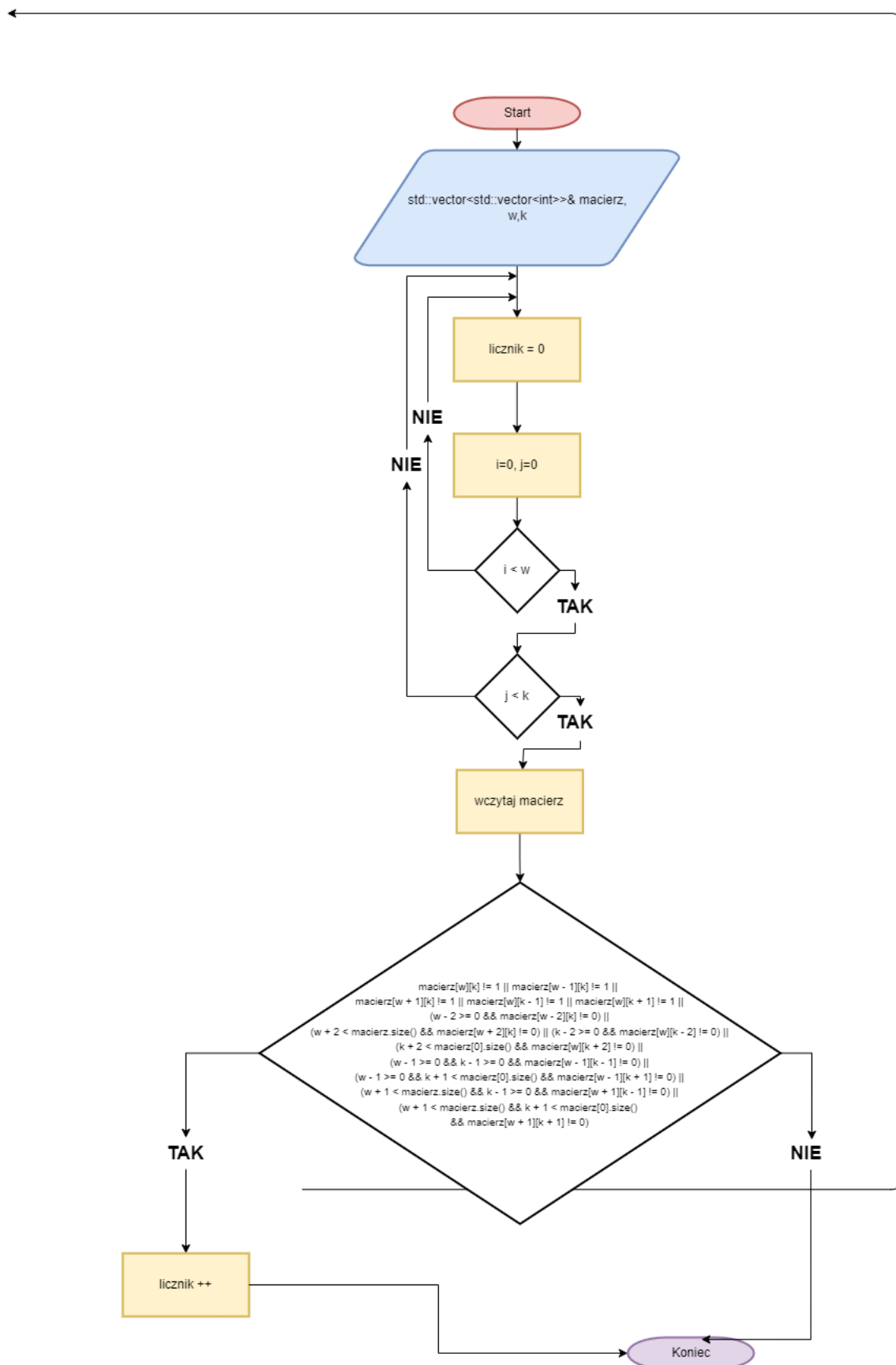
- Liczba znalezionych poprawnych "plusów".

Schemat działania:

1. Należy przeszukiwać całą tablicę lecz z pominięciem jej krawędzi, ponieważ "plus" nie może znajdować się na brzegu.
2. Dla każdego potencjalnego środka "plusa" (1) należy sprawdzić sąsiednie komórki.
3. Aby algorytm działał wydajnie, trzeba unikać wielokrotnego sprawdzania tych samych danych.

2.1.1. Schemat blokowy algorytmu

Algorytm zapisany w postaci schematu blokowego mógłby przedstawiać się następująco:



Grafika 2.1: Schemat blokowy

2.1.3. Algorytm zapisany w pseudokodzie

Poniżej przedstawiam algorytm w formie pseudokodu, który opisuje wydajniejsze rozwiązanie problemu:

```
1      input: std::vector<std::vector<int>>>& macierz // macierz dwuwymiarowa
2          k //ilosc kolumn
3          w //ilosc wierszy
4      licznik := 0
5      i := 0
6      j := 0
7      if i < w
8          while j < k
9              wczytaj macierz
10             endif
11             if(macierz[w][k] != 1 ||
12                 macierz[w - 1][k] != 1 || macierz[w + 1][k] != 1 ||
13                 macierz[w][k - 1] != 1 || macierz[w][k + 1] != 1 ||
14                 (w - 2 >= 0 && macierz[w - 2][k] != 0) ||
15                 (w + 2 < macierz.size() && macierz[w + 2][k] != 0) ||
16                 (k - 2 >= 0 && macierz[w][k - 2] != 0) ||
17                 (k + 2 < macierz[0].size() && macierz[w][k + 2] != 0) ||
18                 (w - 1 >= 0 && k - 1 >= 0 && macierz[w - 1][k - 1] != 0) ||
19                 (w - 1 >= 0 && k + 1 < macierz[0].size() && macierz[w - 1][k + 1] != 0) ||
20                 (w + 1 < macierz.size() && k - 1 >= 0 && macierz[w + 1][k - 1] != 0) ||
21                 (w + 1 < macierz.size() && k + 1 < macierz[0].size() && macierz[w + 1]
22                     [k + 1] != 0)
23             licznik := licznik + 1
24             endif
```

Zapis algorytmu w pseudokodzie, jest etapem pośrednim pomiędzy analizą problemu i opracowaniem algorytmu a samą implementacją w konkretnym języku programowania, która zostanie przedstawiona w kolejnych punktach.

2.1.4 Sprawdzenie poprawności algorytmu

Aby zweryfikować poprawność algorytmu, wykonamy tzw. „ołówkowe” rozwiązanie problemu. Polega ono na ręcznym przeanalizowaniu działania algorytmu dla przykładowej tablicy wejściowej:

7 x 10

```
[0 1 0 0 0 1 0 0 1 0]
[1 1 1 0 0 0 0 1 1 1]
[0 1 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 1 1 1 0 0 0]
[1 1 1 0 0 0 0 0 1 0]
[0 1 0 0 0 0 0 0 0 0]
```

Kroki analizy programu:

1. Iteracja po wszystkich komórkach tablicy, z pominięciem krawędzi.
2. Dla każdej komórki sprawdzono warunki konieczne do uznania jej za środek "plusa".
3. Zidentyfikowano poprawne "plusy" na pozycjach: [2,2], [2,9] oraz [6,2].
4. Łączna liczba "plusów": 3.

Wnioski

Ręczna analiza działania algorytmu potwierdziła poprawność jego implementacji i zgodność z oczekiwaniami wyników.

2.1.5. Teoretyczne oszacowanie złożoności obliczeniowej

Algorytm przetwarza tablicę $M \times N$, iterując przez każdą jej komórkę (z wyłączeniem krawędzi). Dla każdej komórki sprawdzane są warunki utworzenia "plusa", co wymaga:

- Sprawdzenia pięciu komórek (góra, dół, lewo, prawo i środek).
- Sprawdzenia ośmiu komórek wokół "plusa".

Złożoność czasowa algorytmu:

- Iteracja po tablicy wymaga przetworzenia komórek.
- Każda komórka wymaga stałej liczby operacji (sprawdzenie 5+8 komórek).

Stąd całkowita złożoność czasowa wynosi:

Złożoność pamięciowa:

- Algorytm przechowuje tablicę wejściową, co oznacza złożoność pamięciową.
- Dodatkowe zmienne pomocnicze (licznik "plusów") mają również swoją złożoność pamięciową.

Podsumowując, algorytm jest liniowy względem liczby elementów tablicy to znaczy:

- Dla małych tablic (np. 5×5) algorytm wykona stosunkowo niewiele operacji.
- Dla dużych tablic (np. 1000×1000), liczba operacji wzrośnie proporcjonalnie do wielkości tych tablicy.

Złożoność czasowa tego algorytmu wynosi więc $O(M \cdot N)$.

2.1.6 Implementacja algorytmu

Poniżej przedstawiam implementację algorytmu w języku C++, uwzględniającą wszystkie kroki realizacji zadania.

```
#include <iostream>
#include <vector>
#include <fstream>

// Funkcja do sprawdzania, czy dany punkt w tablicy tworzy prawidłowy "plus"
bool czyPoprawnyPlus(const std::vector<std::vector<int>>& macierz, int w, int k) {
    // Sprawdzanie czy centrum "plusa" jest jedynką
    if (macierz[w][k] != 1) return false;

    // Sprawdzanie czy ramiona "plusa" są jedynkami
    if (macierz[w - 1][k] != 1 || macierz[w + 1][k] != 1 ||
        macierz[w][k - 1] != 1 || macierz[w][k + 1] != 1) {
        return false;
    }

    // Sprawdzanie czy wokół "plusa" są zera (lub granice tablicy)
    if ((w - 2 >= 0 && macierz[w - 2][k] != 0) ||
        (w + 2 < macierz.size() && macierz[w + 2][k] != 0) ||
        (k - 2 >= 0 && macierz[w][k - 2] != 0) ||
        (k + 2 < macierz[0].size() && macierz[w][k + 2] != 0) ||
        (w - 1 >= 0 && k - 1 >= 0 && macierz[w - 1][k - 1] != 0) ||
        (w - 1 >= 0 && k + 1 < macierz[0].size() && macierz[w - 1][k + 1] != 0) ||
        (w + 1 < macierz.size() && k - 1 >= 0 && macierz[w + 1][k - 1] != 0) ||
        (w + 1 < macierz.size() && k + 1 < macierz[0].size() && macierz[w + 1][k + 1] != 0)) {
        return false;
    }

    return true;
}

// Funkcja do liczenia liczby prawidłowych "plusów" w tablicy
int liczbaPoprawnychPlusow(const std::vector<std::vector<int>>& macierz) {
    int w = macierz.size();
    int k = macierz[0].size();
    int licznik = 0;

```

Grafika 2.2: Wersja 1

```
// Przechodzi przez wnętrze tablicy (pomija krawędzie)
for (int i = 1; i < w - 1; ++i) {
    for (int j = 1; j < k - 1; ++j) {
        if (czyPoprawnyPlus(macierz, i, j)) {
            ++licznik;
        }
    }
}

return licznik;
}

```

Grafika 2.3: Wersja 1

```

// Funkcja główna programu
void zadanie() {
    // Odczytuje dane z pliku
    std::ifstream plikWejscowy("dane.txt");
    if (!plikWejscowy) {
        std::cerr << "Nie można otworzyć pliku dane.txt!" << std::endl;
        return;
    }

    int w, k;
    plikWejscowy >> w >> k;

    std::vector<std::vector<int>> macierz(w, std::vector<int>(k));

    for (int i = 0; i < w; ++i) {
        for (int j = 0; j < k; ++j) {
            plikWejscowy >> macierz[i][j];
        }
    }

    plikWejscowy.close();

    // Liczenie liczby prawidłowych "plusów"
    int wynik = liczbaPoprawnychPlusow(macierz);

    // Zapis wyniku do pliku
    std::ofstream plikWysciowy("wynik.txt");
    if (!plikWysciowy) {
        std::cerr << "Nie można otworzyć pliku wynik.txt!" << std::endl;
        return;
    }

    plikWysciowy << "Liczba prawidłowych plusów: " << wynik << std::endl;
    plikWysciowy.close();
}

int main() {
    zadanie();
    return 0;
}

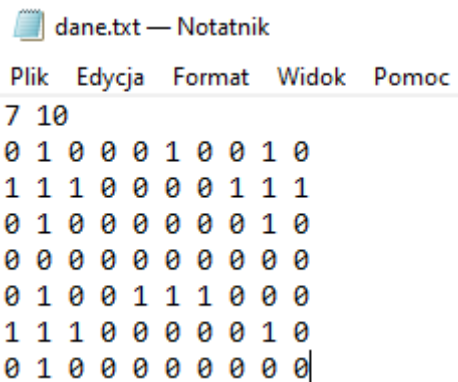
```

Grafika 2.4: Wersja 1

3. Testy i wyniki eksperymentalne

3.1 Przykładowe dane wejściowe i wyjściowe

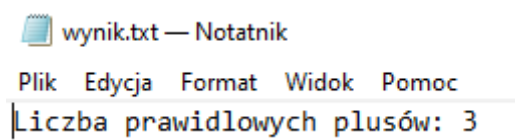
Dane wejściowe:



```
dane.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
7 10
0 1 0 0 0 1 0 0 1 0
1 1 1 0 0 0 0 1 1 1
0 1 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 1 0 0 0
1 1 1 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0
```

Grafika 2.5: Zawartość pliku “dane.txt”

Wynik działania algorytmu:



```
wynik.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
Liczba prawidłowych plusów: 3
```

Grafika 2.6: Zawartość pliku “wynik.txt”

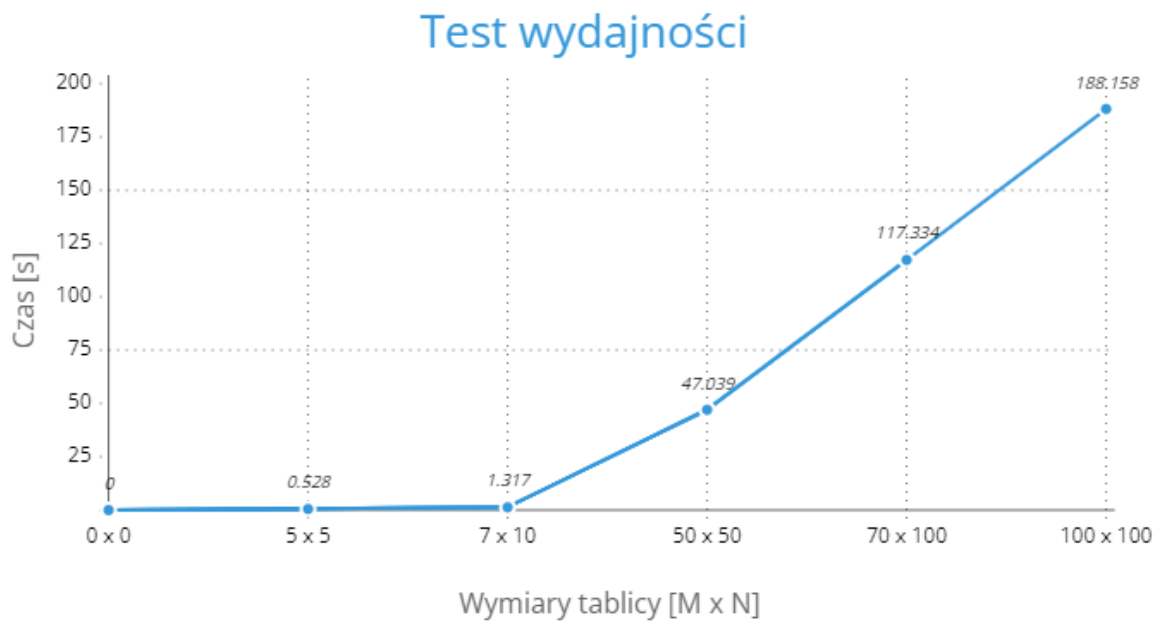
Algorytm poprawnie zidentyfikował 3 "plusy" zgodnie z ręczną analizą w podpunkcie 2.1.4.

3.2 Testy wydajności

Cel: Zmierzenie czasu działania algorytmu dla różnych rozmiarów tablicy.

Metodyka: Generowanie losowych danych wejściowych i pomiar czasu wykonania.

Wyniki:



Grafika 2.7: Wykres wydajności algorytmu

4. Wnioski

Podsumowanie

1. Algorytm został poprawnie zaprojektowany i zaimplementowany. Skutecznie identyfikuje "plusy" w tablicy $M \times N$ zgodnie z treścią zadania.
2. Wydajność algorytmu jest optymalna dla tego typu problemu, z liniową złożonością czasową $O(M \cdot N)$. Pozwala to na przetwarzanie dużych tablic w krótkim czasie.
3. Wyniki testów eksperymentalnych pokazały, że algorytm działa poprawnie i zgodnie z przewidywaniami, zarówno dla przykładowych danych, jak i dużych zestawów testowych.