

<b>SPRAWOZDANIE</b>		<b>Data wykonania:</b> 21.11.2023 – 07.01.2024
<b>Tytuł Mini-Projektu</b>	<b>Wykonała:</b>	<b>Sprawdził:</b>
<i>Mini Projekt 3. Labirynt</i>	<i>Polina Nesterova</i>	<i>dr inż. Konrad Markowski</i>

## Spis treści

Cel projektu .....	2
Rozwiązanie problemu .....	2
Szczegóły implementacyjne .....	4
Sposób wywołania programu .....	9
Wnioski i spostrzeżenia.....	11

## Cel projektu

Projekt ten ma na celu stworzenie generatora labiryntu o rozmiarze otrzymanym od użytkownika (od  $2 \times 2$  do  $10 \times 10$ ). A także stworzenie grafu skierowanego, który przechodziłby przez ten labirynt.

## Rozwiązanie problemu

### 1. Analiza problemu

Projekt rozpoczął się od analizy problemu i określenia listy celów projektu

- algorytm generujący labirynt
- reprezentacja tekstowa
- reprezentacja graficzna
- dodanie wag (losowo wygenerowana liczba zmiennoprzecinkowa)
- graf skierowany, który przechodzi przez labirynt od wejścia do wyjścia

### 2. Planowanie projektu

Określenie modułów i stworzenie listy plików.

- labir.c
  - funkcje do inicjalizacji oraz generowania labiryntu
  - funkcja dla losowego usunięcia ścian w środku labiryntu
  - funkcja dla tekstowej reprezentacji
  - oraz funkcja dla zwolnienia pamięci, którą zajmują labirynt
- labir.h
  - deklaracja funkcji wspomnianych wyżej
  - struktura dla elementów labiryntu
- stos.c
  - implementacja operacji na stosie
- stos.h
  - zawiera deklaracje funkcji operujących na stosie
- graficz.c
  - funkcja do wyświetlenia labiryntu w trybie graficznym
- graficz.h
  - deklaracja funkcji dla wyświetlenia graficznego
- graf.c
  - zawiera funkcje związane z grafem
  - dodawanie wag
  - znajdowanie dróg od wejścia do wyjścia
- main.c
  - główna funkcja programu, w której użytkownik podaje rozmiar labiryntu
  - inicjalizuje labirynt, generuje go, tworzy tekstową i graficzną reprezentację, a następnie wyświetla je

### 3. Algorytm generacji

Po przeczytaniu informacji o różnych typach algorytmów do generowania labiryntu, wybrałem Depth-First Search (DFS).

Krótki opis jego działania

1. Rozpoczęcie od początkowego wierzchołka:
  - Algorytm zaczyna od wybranego wierzchołka startowego.
  - Ten wierzchołek jest oznaczany jako odwiedzony.
2. Przeszukiwanie wierzchołków sąsiednich:
  - Dla danego wierzchołka, algorytm przechodzi do jednego z jego nieodwiedzonych sąsiadów.
  - Jeśli dany wierzchołek nie ma sąsiadów, wraca do poprzedniego wierzchołka (backtracking).
3. Rekurencyjne zagłębianie się:
  - Proces powtarza się rekurencyjnie dla każdego nieodwiedzonego sąsiada.
  - Algorytm 'zagłębia się' w gałąź do samego końca przed powrotem.
4. Oznaczanie wierzchołków:
  - W trakcie przeszukiwania, każdy odwiedzony wierzchołek jest odpowiednio oznaczany.
  - Mogą to być numery kroków, znaczniki czy inne informacje.
5. Zakończenie:
  - Algorytm kończy działanie, gdy odwiedzi wszystkie dostępne wierzchołki lub osiągnie warunek zakończenia.

Więcej informacji można znaleźć na tej stronie

<https://www.algosome.com/articles/maze-generation-depth-first.html>

### 4. Reprezentacja tekstowa i graficzna

Stworzenie funkcji do reprezentacji tekstowej, gdzie ściany i przejścia są odpowiednio oznaczone pewnymi liczbami (więcej informacji o tym dale).

Dla modułu graficznego użyto biblioteki ncurses (biblioteka programistyczna służąca do obsługi interfejsu tekstowego w terminalu).

### 5. Algorytmy grafowe

Wprowadzono algorytmy grafowe w celu analizy dróg prowadzących od wejścia dowyjścia. Dodano wagi do komórek labiryntu, a następnie użyto algorytmu DFS do znalezienia najkrótszych i najdłuższych dróg.

### 6. Optymalizacja i testowanie.

Przeprowadzono testy funkcjonalne, sprawdzające poprawność generowania labiryntu oraz analizy dróg. Dokonano optymalizacji kodu.

## Szczegóły implementacyjne

### Plik labir.c

Wszystkie funkcje i ich tekst nie zostaną tutaj przedstawione, ale tylko najważniejsze części.

```
//inicjalizacja labiryntu o wymiarach size*size
void inicjacja(maze ***labirynt, int size)
{
    int i, j;
    *labirynt = malloc(size * sizeof (maze *));

    for (i = 0; i < size; i++)
    {
        (*labirynt)[i] = malloc(size * sizeof(maze));
        for (j = 0; j < size; j++)
        {
            (*labirynt)[i][j].visited = false;
            (*labirynt)[i][j].top = false;
            (*labirynt)[i][j].bottom = false;
            (*labirynt)[i][j].right = false;
            (*labirynt)[i][j].left = false;
        }
    }
}
```

Funkcja do inicjalizacji labiryntu. Ma na celu przygotowanie struktury labiryntu, inicjalizując wszystkie jej elementy, takie jak ściany, ścieżki, odwiedzone obszary, itp.

```

void generowanie_labiryntu(maze ***labirynt,int size, int **move, int wspol_y, int wspol_x)
{
    int punkt_y = wspol_y;
    int punkt_x = wspol_x;
    dodaj_wagi(labirynt, size);
    (*labirynt)[punkt_y][punkt_x].visited = true;
    push_stack(move, punkt_x, punkt_y, size*size);

    while( top_stack(move, size*size) != -1)
    {
        while( usun_sciane(labirynt, size, move, &punkt_x, &punkt_y) == -1 )
        {
            pop_stack(move, size*size);
            int indeks = top_stack(move, size*size);
            if (indeks == -1)
            {
                return;
            }
            punkt_y = move[indeks][0];
            punkt_x = move[indeks][1];
        }
        if (punkt_x == wspol_x && punkt_y == wspol_y)
        {
            return;
        }
    }
}

```

Funkcja ma na celu wygenerowanie labiryntu o określonym rozmiarze, używając algorytmu DFS do losowego przeszukiwania i tworzenia ścieżek w strukturze labiryntu.

Następna funkcja **wysw\_tekstowe** wykonuje następujące zadania

- Najpierw wypełnienie liczbą 4
- Potem wypełnienie ścian 2
- Ustawienie każdej 2iej kratki (od 0 do  $2n + 1$ ) na 1
- Ustawienie komórki powyżej wierzchołków (po analizie tego jaka jest zawartość labiryntu)
- Ustawienie komórki z prawej strony od wierzchołka
- Losowe usunięcie ścian
- Zerowanie początkowej i końcowej ściany
- Koregowanie ścian bocznych, żeby były stworzone wyłącznie z 2
- Oraz wypisanie labiryntu w postaci liczb

Podsumowując –

- 0, 1 - ściana pusta lub nie
- 2 - ściana zewnętrzna
- 3 - wierzchołek
- 4 - wypełnienie dla debugowania

Funkcja **usun\_sciane** pełni ważną rolę w algorytmie generowania labiryntu. Jej zadaniem jest usuwanie ściany pomiędzy komórkami, tworząc losowe przejścia i umożliwiając rozwinięcie struktury labiryntu.

- Sprawdza dostępność kierunków (góra, dół, prawo, lewo) w aktualnej komórce labiryntu.
- Losowo wybiera jeden z dostępnych kierunków.
- Usuwa ścianę pomiędzy aktualną komórką a wybranym sąsiadem, oznaczając je odpowiednio jako odwiedzone.
- Dodaje współrzędne nowego punktu do stosu ruchów.

Plik **stos.c**

```
int pop_stack(int **stos_z_ruchami, int rozmiar) {
    int i = 0;
    while (i < rozmiar && stos_z_ruchami[i][0] != -1 && stos_z_ruchami[i][1] != -1) {
        i++;
    }
    if (i == 0) {
        //Stos pusty
        return -1;
    } else {
        stos_z_ruchami[i - 1][0] = -1;
        stos_z_ruchami[i - 1][1] = -1;
        return 0;
    }
}
```

Usuwa element z wierzchołka stosu ruchów. Szuka ostatniego elementu na stosie (elementu o współrzędnych różnych od -1). Jeśli stos nie jest pusty, usuwa ostatni element ze stosu.

```
int top_stack(int **stos_z_ruchami, int rozmiar) {
    int i = 0;
    while (i < rozmiar && stos_z_ruchami[i][0] != -1 && stos_z_ruchami[i][1] != -1) {
        i++;
    }
    if (i == 0) {
        //Stos pusty
        return -1;
    } else {
        return i - 1;
    }
}
```

Zwraca indeks ostatniego elementu na stosie ruchów. Szuka ostatniego elementu na stosie (elementu o współrzędnych różnych od -1). Jeśli stos nie jest pusty, zwraca indeks ostatniego elementu.

```

int push_stack(int **stos_z_ruchami, int x, int y, int rozmiar) {
    int i = 0;
    while (i < rozmiar && stos_z_ruchami[i][0] != -1 && stos_z_ruchami[i][1] != -1) {
        i++;
    }
    if (i >= rozmiar) {
        //Stos przepełniony
        return -1;
    }
    stos_z_ruchami[i][0] = y;
    stos_z_ruchami[i][1] = x;
    return 0;
}

```

Dodaje element na stos ruchów używanego w algorytmie generowania labiryntu. Szuka pierwszego wolnego miejsca na stosie (elementu o współrzędnych -1).

Jeśli stos nie jest przepełniony, dodaje współrzędne nowego punktu na stos.

Plik graficz.c

```

void wyswietl_labirynt_graficznie(int **tablica_wys_lab, int rozmiar) {
    initscr(); // Inicjalizacja ncurses
    start_color(); // Włączenie kolorów
    init_pair(1, COLOR_BLACK, COLOR_WHITE); // Kolor dla ścian
    init_pair(2, COLOR_BLACK, COLOR_GREEN); // Kolor dla przejść

    for (int i = 0; i < 2 * rozmiar + 1; i++) {
        for (int j = 0; j < 2 * rozmiar + 1; j++) {
            if (tablica_wys_lab[i][j] == 1 || tablica_wys_lab[i][j] == 2) {
                attron(COLOR_PAIR(1));
                printw("%c ", 178);
            } else if (tablica_wys_lab[i][j] == 0 || tablica_wys_lab[i][j] == 3) {
                attron(COLOR_PAIR(2));
                printw("%c ", 176);
            } else if (tablica_wys_lab[i][j] == 4) {
                printw("w");
            }
        }
        printw("\n");
    }

    refresh(); // Odświeżenie ekranu
    getch(); // Oczekiwanie na klawisz
    endwin(); // Zakończenie ncurses
}

```

Funkcja **wyswietl\_labirynt\_graficznie** używa biblioteki ncurses do wyświetlenia labiryntu w trybie graficznym.

- Inicjalizuje bibliotekę ncurses, umożliwiając obsługę terminala w trybie graficznym.
- Ustawia kolory dla ścian i przejść w labiryncie.

- Przechodzi przez każdy element tablicy labiryntu, wybierając odpowiedni kolor w zależności od wartości.
- Wykorzystuje funkcje ncurses do wyświetlenia elementów labiryntu na ekranie.
- Oczekuje na naciśnięcie klawisza, po czym kończy działanie, przywracając normalny tryb terminala.

### Ważne informacje.

Funkcja wymaga zainstalowanej biblioteki ncurses. W terminalu można to zrobić za pomocą komendy

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

Plik graf.c

```
// Funkcja dodająca wagę do komórek, które są przejściami
void dodaj_wagi(maze ***labirynt, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if ((*labirynt)[i][j].top) {
                (*labirynt)[i][j].weight = (float)(rand() % 11);
            }
        }
    }
}
```

Przypisuje losowe wagi przejściom w labiryncie. Przechodzi przez każdą komórkę labiryntu. Jeśli dany kierunek (top) jest przejściem (brak ściany), przypisuje losową wagę z zakresu od 0 do 10.

Funkcja **dfs** (Depth-First Search) jest algorytmem przeszukiwania grafu w głąb. W kontekście projektu służy do rekurencyjnego szukania dróg w labiryncie.

- Funkcja przeszukuje labirynt rekurencyjnie, przechodząc przez sąsiednie komórki.
- Oznacza komórki jako odwiedzone i aktualizuje wagę ścieżki.
- Jeśli dotrze do celu (ostatniej komórki), aktualizuje minimalną i maksymalną wagę.
- Rekurencyjnie wywołuje się dla każdego sąsiada, pomijając komórki z ścianami

Funkcja **znajdz\_drogi** używa algorytmu DFS do znalezienia dróg od wejścia do wyjścia w labiryncie, obliczając jednocześnie sumy wag dla każdej z nich.

- Alokuje dynamicznie tablicę visited do śledzenia odwiedzonych komórek.
- Inicjalizuje minimalną wagę na FLT\_MAX, a maksymalną na 0.
- Wywołuje rekurencyjną funkcję dfs dla punktu startowego (0,0).
- Wypisuje informacje o ilości znalezionych dróg, najkrótszej i najdłuższej drodze (sumie wag).
- Zwolnienie zaalokowanej pamięci.

Plik main.c



Plik **main.c** zawiera funkcję główną programu, która inicjalizuje labirynt, generuje go, wizualizuje go w formie tekstowej i graficznej, a następnie analizuje dróżki w labiryncie

- Użytkownik jest pytany o rozmiar labiryntu, a następnie inicjalizowany i generowany jest labirynt.
- Wykorzystywane są struktury danych i funkcje z plików nagłówkowych, takie jak `stos` czy algorytmy grafowe.
- Labirynt jest wizualizowany w formie tekstowej i graficznej za pomocą `ncurses`.
- Wywoływane są funkcje analizujące dróżki w labiryncie, takie jak `znajdz_drogi`.
- Na końcu program zwalnia zaalokowaną pamięć.

## Sposób wywołania programu

Aby zobaczyć działanie programu, użytkownik powinien skompilować program w następujący sposób.

```
gcc main.c stos.c labir.c graficz.c graf.c -lncurses -lm
```


Aby zobaczyć graficzną reprezentację labiryntu, należy zainstalować bibliotekę `ncurses`. W Terminalu jest to możliwe za pomocą komendy

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

### Przykładowa kompilacja

```
gcc main.c stos.c labir.c graficz.c graf.c -lncurses -lm
```

Użytkownik otrzymuje graficzną reprezentację.

 nesterop@DESKTOP-OA2OQN4: /mnt/d/PINF/123



Kliknąć na dowolny przycisk.

I następnie można zobaczyć reprezentację tekstową oraz informację o drogach.

```

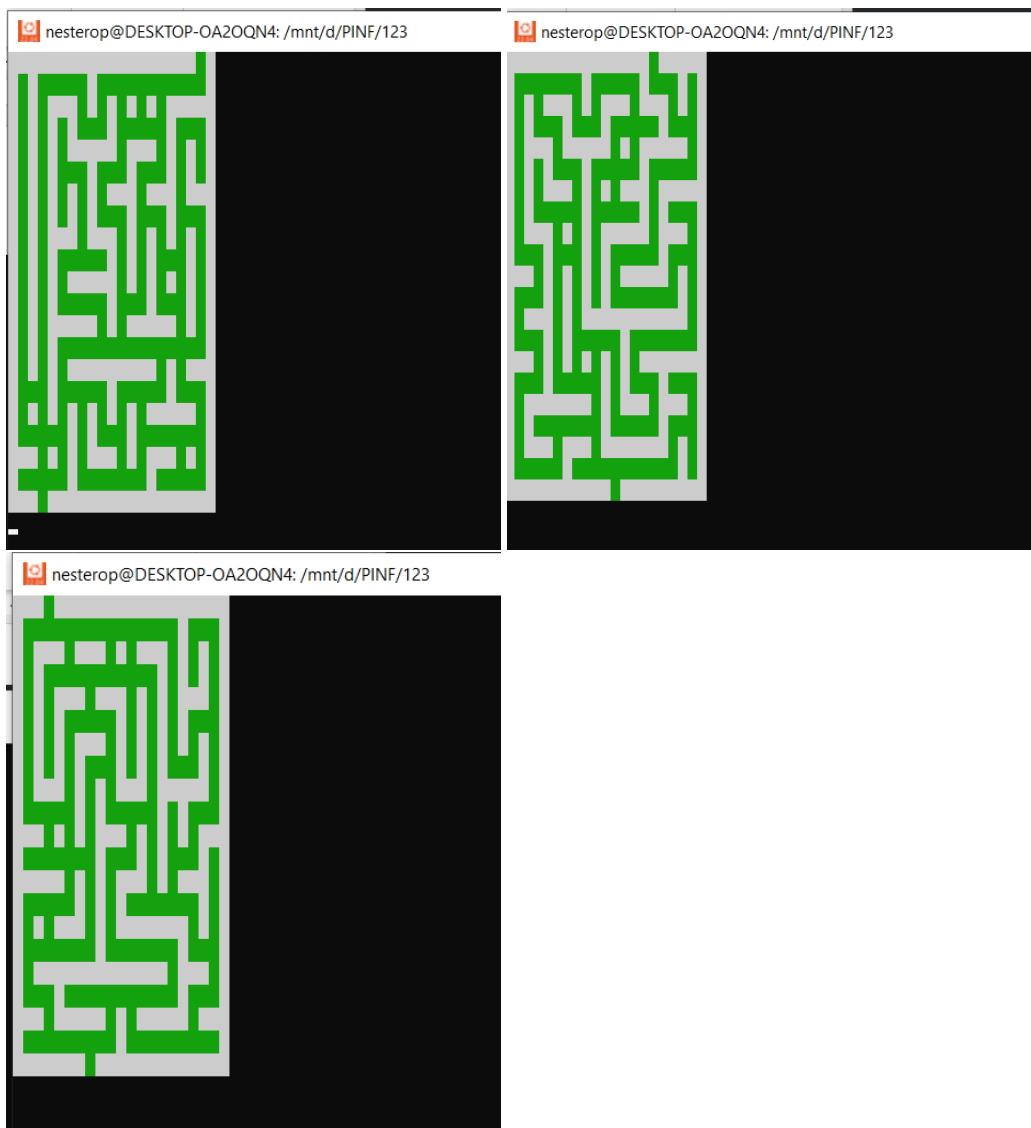
nesterop@DESKTOP-OA2OQN4:/mnt/d/PINF/123$ gcc main.c stos.c labir.c graficz.c graf.c -lncurses -lm
nesterop@DESKTOP-OA2OQN4:/mnt/d/PINF/123$ ./a.out
Podaj rozmiar labiryntu: 3

2 2 2 0 2 2 2
2 3 0 3 1 3 2
2 0 2 1 2 0 2
2 3 1 3 0 3 2
2 0 2 0 2 0 2
2 3 0 3 0 3 2
2 2 2 0 2 2 2
Ilość dróg prowadzących od wejścia do wyjścia: 12
Najkrótsza droga (suma wag): 10.00
Najdłuższa droga (suma wag): 28.00
nesterop@DESKTOP-OA2OQN4:/mnt/d/PINF/123$ ./a.out
Podaj rozmiar labiryntu: 2

2 0 2 2 2
2 3 0 3 2
2 0 2 0 2
2 3 0 3 2
2 0 2 2 2
Ilość dróg prowadzących od wejścia do wyjścia: 2
Najkrótsza droga (suma wag): 10.00
Najdłuższa droga (suma wag): 11.00
nesterop@DESKTOP-OA2OQN4:/mnt/d/PINF/123$

```

Jak wygląda labirynt o większych wymiarach.



Czasami, podczas sprawdzania działania programu z labiryntem o dużych wymiarach, program może się zawiesić. Aby rozwiązać ten problem, wystarczy nacisnąć "Ctrl + C".

## Wnioski i spostrzeżenia

Za główne osiągnięcie w tym projekcie uważam to, że udało mi się stworzyć program, który w przeciwieństwie do wszystkich poprzednich wersji, generuje labirynt całkowicie poprawnie. Zawsze jest tylko jedno wejście i jedno wyjście. I nigdy nie ma żadnych zamkniętych przestrzeni, do których graf skierowany nie może się dostać. Jestem również bardzo zadowolona ze sposobu, w jaki udało się stworzyć graficzną reprezentację labiryntu, a nie tylko # dla ściany i ' ' dla przejścia. Wadą wykonanej pracy jest niedoskonałość funkcji odpowiedzialnych za graf skierowany oraz fakt, że program czasami zawiesza się podczas pracy z dużymi labiryntami.