Sprawozdanie z miniprojektu 4

Mateusz Kamieniecki

SPRAWOZDANIE	**Data wykonania: 28.12.2025**
Tytuł miniprojektu:	*Labirynt*
Wykonał:	*Mateusz Kamieniecki*
Sprawdził:	*dr inż. Konrad Markowski*

Spis treści

1 Cel Projektu

Celem projektu było stworzenie lowosego labiryntu w którym można dostać się do każdego wierzchołka, zrobioć z niego graf, znaleźć najkrótszą ścieżkę między początkiem i końcem labiryntu oraz znaleźć pozostałe ścieżki.

2 Rozwiązanie Problemu

2.1 Tworzenie losowego labiryntu

Aby stworzyć losowy labirynt jest kilka różnych algorytmów. Ja do tego problemu wybrałęm generację za pomocą dfs ale krótko opiszę na czym polegają. Po zakończeniu algorytmu DFS (Depth-First Search) losowo są niszczone ściany, aby stworzyć więcej ścieżek w labiryncie.

2.1.1 Algorytm DFS

Algorytm DFS polega na tym, że zaczynamy od losowego punktu i idziemy w losowym kierunku aż do momentu, gdy nie możemy iść dalej. Wtedy wracamy się do ostatniego punktu, gdzie mieliśmy jeszcze nieodwiedzone sąsiednie komórki i kontynuujemy. Powtarzamy ten proces aż odwiedzimy wszystkie komórki. Ten algorytm ma tendencję do tworzenia długich, krętych ścieżek.

2.1.2 Algorytm Kruskala

Algorytm Kruskala polega na traktowaniu każdej ściany labiryntu jako krawędzi w grafie. Losowo wybieramy krawędzie i usuwamy je, jeśli usunięcie nie spowoduje powstania cyklu w grafie. Powtarzamy ten proces, aż wszystkie komórki będą połączone. Ten algorytm tworzy labirynty, które mają dużo krótkich ścieżek i rozgałęzień.

2.1.3 Algorytm Wilsona

Algorytm Wilsona polega na tworzeniu losowych ścieżek z nieodwiedzonych komórek do już odwiedzonych komórek. Gdy ścieżka napotka odwiedzoną komórkę, łączymy ją z labiryntem. Powtarzamy ten proces, aż wszystkie komórki będą odwiedzone. Ten algorytm tworzy labirynty z równomiernym rozkładem ścieżek.

2.2 Znajdywanie Najkrótszej Ścieżki

Początkowo używałem tego samego algorytmu DFS do znajdywania wszystkich ścieżek oraz najkrótszej ścieżki. Jednak DFS nie jest optymalnym wyborem do znajdywania najkrótszej ścieżki w grafie o równych wagach krawędzi, ponieważ może prowadzić do eksploracji długich ścieżek zanim znajdzie krótszą. Zamiast tego, lepszym wyborem jest algorytm BFS (Breadth-First Search). BFS eksploruje wszystkie sąsiednie wierzchołki na danym poziomie przed przejściem do następnego poziomu, co gwarantuje znalezienie najkrótszej ścieżki w grafie o równych wagach krawędzi.

2.3 Znajdywanie Pozostałych Ścieżek

Do znajdywania wszystkich ścieżek między dwoma punktami w labiryncie użyłem algorytmu DFS (Depth-First Search). DFS jest odpowiedni do tego zadania, ponieważ pozwala na eksplorację wszystkich możliwych ścieżek poprzez rekurencyjne przechodzenie przez każdy możliwy kierunek z aktualnego wierzchołka, aż do momentu dotarcia do punktu docelowego lub wyczerpania wszystkich opcji.

Ponieważ liczba wszystkich możliwych ścieżek może być bardzo duża, postanowiłem wprowadzić limit na maksymalną liczbę ścieżek do znalezienia. W mojej implementacji ustawiłem limit na 500 ścieżek.

3 Szczegóły Implementacji

3.1 Reprezentacja Labiryntu w Ascii

Ciężko jest przedstawić ściany labiryntu oraz pole w jednym znaku dlatego postanowiłem rozszerzyć labirynt, aby oddzielne komórki miały swoje ściany. Labirynt o rozmiarze NxN jest reprezentowany jako tablica o rozmiarze (2N+1)x(2N+1).

Ściany są reprezentowane przez znaki kwadratowe, pola są reprezentowane jako liczby, a ścieżki pomiędzy jako kreski.

Listing 1: Funkcja drukująca labirynt

```
void print_labyrynth(int n, int t[(2 * n) + 1][(2 * n) + 1]) {
    for (int i = 0; i < (2 * n) + 1; i++) {
        for (int j = 0; j < (2 * n) + 1; j++) {
            if (t[i][j] == 1) {
                // Sciana
                printf("");
            } else if (i % 2 == 1 && j % 2 == 1) {
                // Węzeł (pokój)
                int node_i = (i - 1) / 2;
                int node_j = (j - 1) / 2;
                int node_id = node_i * n + node_j;
                if (node_id >= 100) {
                    printf(""); // Dla dużych liczb używamy ".."
                } else {
                    printf("%2d", node_id);
                }
            } else if (i % 2 == 1 && j % 2 == 0) {
                // Poziomy korytarz
                printf("");
            } else if (i % 2 == 0 && j % 2 == 1) {
                // Pionowy korytarz
                printf("");
            } else {
                // Narożnik
                printf("");
            }
        }
        printf("\n");
    }
}
```

(te znaki ASCII zostały usunięte, ponieważ nie są poprawnie wyświetlane w tym formacie w LaTeX)

3.2 Wybieranie kierunku

Kierunek wybierany jest losowo z tablicy kierunków. Poniżej znajduje się fragment kodu definiujący przykładową tablicę kierunków:

Listing 2: Tablica kierunków

```
int directions[4][2] = {{0, 2}, {2, 0}, {0, -2}, {-2, 0}};
```

w ten sposób można łatwo iterować po tablicy i wybierać losowy kierunek.

3.3 Reprezentacja grafu

Postanowiłem użyć listy sąsiedztwa, którą napisałem w poprzednim projekcie. Lista sąsiedzctwa jest według mnie najwygodniejszą reprezentacją grafu do tego typu zadań.

3.4 Kolejka w BFS

Do implementacji kolejki w algorytmie BFS potrzebna jest kolejka, aby przechowywać wierzchołki do odwiedzenia. W mojej implementacji użyłem prostej kolejki, w której trzymam początek i koniec kolejki oraz tablicę do przechowywania elementów.

Listing 3: Inicjalizacja kolejki

```
int *queue = malloc(n * n * sizeof(int));
int front = 0, rear = 0;
```

4 Sposób wykorzystania programu

Program jest kompilowany za pomocą GNU make. Aby skompilować program, należy wykonać polecenie `make` w katalogu głównym projektu. Spowoduje to utworzenie pliku wykonywalnego o nazwie `main`.

Aby wyczyścić pliki obiektowe i plik wykonywalny, należy wykonać polecenie `make clean`.

W przypadku braku zainstalowanego GNU make, program można skompilować ręcznie za pomocą następującego polecenia:

Listing 4: Ręczna kompilacja programu

```
gcc -Wall -Wextra -std=c11 -o labyrinth 343333.c graph.c
```

Program można uruchomić wykonując polecenie: `./labyrinth` program poprosi o podanie rozmiaru labiryntu. Potem wypisze labirynt, połączenia w grafie, najkrótszą ścieżkę oraz pozostałe ścieżki.

5 Wnioski

Podczas realizacji projektu nauczyłem się jak tworzyć losowe labirynty oraz jak reprezentować je jako grafy. Dodatkowo poznałem jak implementować algorytmy DFS oraz BFS w C.

6 Dodatek

Gdyby MS Teams znowu robiło problemy z załącznikami, kod źródłowy projektu jest dostępny na GitHub pod adresem: `https://github.com/DoggoGood/pi_miniproj4`