SPRAWOZDANIE		Data wykonania: 06.01.2024
Tytuł Mini-Projektu	**Wykonał:**	**Sprawdził:**
Mini Projekt 3	*Kacper Koziara*	*dr inż. Konrad Markowski*

Cel projektu

Napisać program który:

- ma pobierać od użytkownika dwie liczby całkowite dodatnie i nieparzyste, które oznaczają szerokość i wysokość labiryntu. Jeśli użytkownik poda niepoprawne dane, program ma zwrócić błąd.
- ma generować losowy labirynt o podanych wymiarach, używając algorytmu przeszukiwania w głąb (DFS), który losowo wybiera sąsiednie komórki i łączy je, tworząc ścieżki. Program ma zapewnić, że labirynt ma ściany na brzegach i nie ma luk w środku.
- ma wyświetlać labirynt na ekranie, używając znaków ASCII do reprezentowania ścian, ścieżek, punktu startowego i końcowego. Program ma losowo wybierać punkt startowy i końcowy z brzegów labiryntu.
- ma tworzyć graf na podstawie labiryntu, używając struktury grafu, która zawiera liczbę wierzchołków i tablicę list sąsiedztwa. Program ma traktować każdą komórkę labiryntu jako wierzchołek grafu i łączyć je krawędziami, jeśli sąsiadują ze sobą i nie ma między nimi ściany.
- Program ma znajdować najkrótszą ścieżkę w grafie.
- Program ma zaznaczać najkrótszą ścieżkę w labiryncie. Program ma ponownie wyświetlać labirynt na ekranie z zaznaczoną ścieżką.

Rozwiązanie problemu

- Najpierw zrozumiałem, jaki jest problem i jakie są jego założenia. Problem polegał na tym, żeby napisać kod, który generuje, wyświetla i rozwiązuje labirynt. Założenia były takie, że labirynt ma być prostokątny, o wymiarach podanych przez użytkownika, i ma mieć jeden punkt startowy i jeden końcowy. Labirynt ma być reprezentowany jako graf, a najkrótsza ścieżka ma być zaznaczona w labiryncie.
- Potem zaplanowałem, jak podzielić problem na mniejsze i prostsze pod problemy. Zdecydowałem, że mój kod ma składać się z kilku funkcji, które wykonują określone zadania, takie jak:
    - generowanie labiryntu za pomocą algorytmu DFS
    - tworzenie grafu na podstawie labiryntu
    - znajdowanie najkrótszej ścieżki w grafie za pomocą algorytmu Dijkstry
    - zaznaczanie ścieżki w labiryncie
    - wyświetlanie labiryntu na ekranie
    - zwalnianie pamięci
    - obsługa danych wejściowych od użytkownika
- Potem zacząłem pisać kod, zaczynając od najważniejszych i najtrudniejszych funkcji, takich jak `generate_maze` i `dijkstra`. Używałem różnych zmiennych, struktur danych, instrukcji sterujących i funkcji, które były potrzebne do realizacji logiki programu.
- Potem sprawdziłem, czy mój kod działa poprawnie i spełnia wymagania zadania. Napotkałem kilka błędów i ostrzeżeń, takich jak brakujące nawiasy klamrowe, niezgodność typów lub niepoprawne nazwy zmiennych. Wykrywałem i naprawiałem te błędy i ostrzeżenia, używając informacji zwrotnej z kompilatora i debuggera. Sprawdzałem też, czy mój kod obsługuje różne dane wejściowe od użytkownika i czy sprawdza ich poprawność.

Szczegóły implementacyjne

Szczegóły implementacyjne tego programu są następujące:

- Program składa się z kilku funkcji, które wykonują określone zadania, takie jak:
    - `generate_maze` - generuje labirynt za pomocą algorytmu przeszukiwania w głąb (DFS), który losowo wybiera sąsiednie komórki i łączy je, tworząc ścieżki. Funkcja ta używa stosu do przechowywania aktualnej ścieżki i tablicy do przechowywania stanu komórek (odwiedzone lub nieodwiedzone).
    - `print_maze` - wyświetla labirynt na ekranie, używając znaków ASCII do reprezentowania ścian, ścieżek, punktu startowego, końcowego i zaznaczonego. Funkcja ta używa pętli for do iterowania po tablicy labiryntu i funkcji printf do wyświetlania znaków.
    - `generate_graph` - tworzy graf na podstawie labiryntu, używając struktury graph, która zawiera liczbę wierzchołków i tablicę list sąsiedztwa. Funkcja ta używa pętli for do iterowania po tablicy labiryntu i funkcji malloc do alokowania pamięci dla list sąsiedztwa.
    - `dijkstra` - znajduje najkrótszą ścieżkę w grafie za pomocą algorytmu Dijkstry, który używa kolejki priorytetowej do przechowywania wierzchołków według ich odległości od źródła. Funkcja ta używa tablic do przechowywania odwiedzonych wierzchołków, odległości i poprzedników. Funkcja ta zwraca odległość do celu lub 999, jeśli nie ma rozwiązania.
    - `mark_path` - zaznacza najkrótszą ścieżkę w labiryncie za pomocą znaku MARK, używając tablicy prev do odtworzenia ścieżki od celu do źródła. Funkcja ta używa pętli while do iterowania po tablicy prev i warunku if do sprawdzenia, czy komórka nie jest startem ani końcem.
    - `free_maze` - zwalnia pamięć alokowaną dla tablicy labiryntu, używając pętli for do iterowania po wierszach labiryntu i funkcji free do zwalniania pamięci dla każdego wiersza.
    - `init_arrays` - inicjalizuje tablice pomocnicze, używając pętli for do iterowania po tablicach i przypisując im odpowiednie wartości. Funkcja ta używa funkcji malloc do alokowania pamięci dla tablic i stałych do określenia wartości początkowych.
    - `main` - główna funkcja programu, która pobiera dane od użytkownika, generuje, wyświetla i rozwiązuje labirynt, zaznacza i wyświetla najkrótszą ścieżkę, i zwalnia pamięć. Funkcja ta używa funkcji scanf do pobierania danych od użytkownika, funkcji rand do generowania liczb losowych, funkcji srand do ustawiania ziarna generatora, i funkcji printf do wyświetlania danych na ekranie. Funkcja ta używa też warunków if do sprawdzania poprawności danych wejściowych i istnienia rozwiązania.

Oto kod programu:

```
#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#define WALL '#' // znak oznaczający ścianę

#define PATH '.' // znak oznaczający ścieżkę

#define START 'S' // znak oznaczający start

#define END 'E' // znak oznaczający koniec

#define MARK '*' // znak oznaczający znacznik

typedef struct point{ // struktura przechowująca
współrzędne punktu

    int x;

    int y;



}

point;

point start; // zmienna przechowująca punkt startowy

point end; // zmienna przechowująca punkt końcowy

int is_valid(int x,int y,int width,int height){ //
funkcja sprawdzająca, czy punkt jest w granicach
labiryntu

    return x>=0&&x<width&&y>=0&&y<height;



}

    int count_wall_neighbors(char**maze,point p,int
width,int height){ // funkcja licząca liczbę
sąsiadów będących ścianami dla danego punktu

        int count=0;
```

```
        int dx[]={-1,0,1,0}; // tablica
przechowująca przesunięcia w osi x

        int dy[]={0,-1,0,1}; // tablica
przechowująca przesunięcia w osi y

        for(int i=0;i<4;i++){ // pętla po czterech
kierunkach

            int nx=p.x+dx[i]; // nowa współrzędna x

            int ny=p.y+dy[i]; // nowa współrzędna y


if(is_valid(nx,ny,width,height)&&maze[ny][nx]==WALL)
{ // jeśli punkt jest w granicach i jest ścianą

                count++; // zwiększ licznik



        }

        }

        return count;} // zwróć liczbę sąsiadów
będących ścianami

        void generate_maze(char**maze,point p,int
width,int height){ // funkcja generująca labirynt za
pomocą algorytmu DFS

            maze[p.y][p.x]=PATH; // oznacz punkt
jako ścieżkę

            int order[]={0,1,2,3}; // tablica
przechowująca kolejność odwiedzania sąsiadów

            for(int i=0;i<4;i++){ // pętla losująca
kolejność

                int j=rand()%4; // losowy indeks

                int temp=order[i]; // zamiana
miejscami

                order[i]=order[j];

                order[j]=temp;
```

```
            }

            int dx[]={-1,0,1,0}; // tablica
przechowująca przesunięcia w osi x

            int dy[]={0,-1,0,1}; // tablica
przechowująca przesunięcia w osi y

            for(int i=0;i<4;i++){ // pętla po
czterech kierunkach

                int nx=p.x+dx[order[i]]; // nowa
współrzędna x

                int ny=p.y+dy[order[i]]; // nowa
współrzędna y

                point np={nx,ny}; // nowy punkt


if(is_valid(nx,ny,width,height)&&count_wall_neighbor
s(maze,np,width,height)>=3){ // jeśli punkt jest w
granicach i ma co najmniej trzech sąsiadów będących
ścianami


generate_maze(maze,np,width,height); //
rekurencyjnie odwiedź punkt



                }



            }



        } // zakończ funkcję



       void print_maze(char**maze,int width,int
height){ // funkcja wyświetlająca labirynt na
ekranie
```

```
            for(int y=0;y<height;y++){ // pętla po
wierszach
                for(int x=0;x<width;x++){ // pętla
po kolumnach
                    printf("%c ",maze[y][x]); //
wydrukuj znak


                }
                printf("\n"); // przejdź do nowej
linii


            }


        }
        void free_maze(char**maze,int height){ //
funkcja zwalniająca pamięć labiryntu
            for(int y=0;y<height;y++){ // pętla po
wierszach
                free(maze[y]); // zwolnij pamięć
wiersza


            }
            free(maze); // zwolnij pamięć tablicy


        }
typedef struct node { // struktura przechowująca
węzeł listy sąsiedztwa
    int x; // współrzędna x węzła
    int y; // współrzędna y węzła
```

```
    struct node* next; // wskaźnik na następny węzeł
w liście



} node;

typedef struct graph { // struktura przechowująca
graf

    int V; // liczba wierzchołków

    node** adj; // tablica list sąsiedztwa



} graph;

graph* create_graph(int V) { // funkcja tworząca
graf

    graph* g = (graph*)malloc(sizeof(graph)); //
alokacja pamięci dla grafu

    g->V = V; // ustawienie liczby wierzchołków

    g->adj = (node**)malloc(V * sizeof(node*)); //
alokacja pamięci dla tablicy list sąsiedztwa

    for (int i = 0; i < V; i++) { // pętla po
wierzchołkach

        g->adj[i] = NULL; // ustawienie listy
sąsiedztwa na pustą



    }

    return g; // zwrócenie grafu



}

void add_edge(graph* g, int u, int v) { // funkcja
dodająca krawędź skierowaną od wierzchołka u do
wierzchołka v
```

```
    node* new_node = (node*)malloc(sizeof(node)); //
alokacja pamięci dla nowego węzła

    new_node->x = v / g->V; // obliczenie
współrzędnej x węzła

    new_node->y = v % g->V; // obliczenie
współrzędnej y węzła

    new_node->next = g->adj[u]; // ustawienie
wskaźnika na następny węzeł na głowę listy
sąsiedztwa u

    g->adj[u] = new_node; // ustawienie nowego węzła
na głowę listy sąsiedztwa u



} // zakończ funkcję



graph* generate_graph(char** maze, int width, int
height) { // funkcja tworząca graf na podstawie
labiryntu

    graph* g = create_graph(width * height); //
utworzenie grafu o liczbie wierzchołków równej
iloczynowi szerokości i wysokości labiryntu

    for (int i = 0; i < height; i++) { // pętla po
wierszach labiryntu

        for (int j = 0; j < width; j++) { // pętla
po kolumnach labiryntu

            if (maze[i][j] == PATH || maze[i][j] ==
START || maze[i][j] == END) { // jeśli komórka
labiryntu jest ścieżką, startem lub końcem

                int u = i * width + j; // obliczenie
indeksu wierzchołka odpowiadającego komórce

                if (i > 0 && (maze[i-1][j] == PATH
|| maze[i-1][j] == START || maze[i-1][j] == END)) {
// jeśli komórka nad jest ścieżką, startem lub
końcem

                    int v = (i-1) * width + j; //
obliczenie indeksu wierzchołka odpowiadającego
komórce nad
```

```
                    add_edge(g, u, v); // dodanie
krawędzi od u do v



                }

                if (i < height - 1 && (maze[i+1][j]
== PATH || maze[i+1][j] == START || maze[i+1][j] ==
END)) { // jeśli komórka pod jest ścieżką, startem
lub końcem

                    int v = (i+1) * width + j; //
obliczenie indeksu wierzchołka odpowiadającego
komórce pod

                    add_edge(g, u, v); // dodanie
krawędzi od u do v



                }

                if (j > 0 && (maze[i][j-1] == PATH
|| maze[i][j-1] == START || maze[i][j-1] == END)) {
// jeśli komórka po lewej jest ścieżką, startem lub
końcem

                    int v = i * width + (j-1); //
obliczenie indeksu wierzchołka odpowiadającego
komórce po lewej

                    add_edge(g, u, v); // dodanie
krawędzi od u do v



                }

                if (j < width - 1 && (maze[i][j+1]
== PATH || maze[i][j+1] == START || maze[i][j+1] ==
END)) { // jeśli komórka po prawej jest ścieżką,
startem lub końcem

                    int v = i * width + (j+1); //
obliczenie indeksu wierzchołka odpowiadającego
komórce po prawej

                    add_edge(g, u, v); // dodanie
krawędzi od u do v
```

```
                }

            }

        }

    }

    return g; // zwrócenie grafu

}

int* visited; // tablica przechowująca informację,
czy wierzchołek został odwiedzony

int* dist; // tablica przechowująca odległość od
wierzchołka startowego

int* prev; // tablica przechowująca poprzednika
wierzchołka na najkrótszej ścieżce

void init_arrays(int V) { // funkcja inicjalizująca
tablice

    visited = (int*)malloc(V * sizeof(int)); //
alokacja pamięci dla tablicy visited

    dist = (int*)malloc(V * sizeof(int)); //
alokacja pamięci dla tablicy dist

    prev = (int*)malloc(V * sizeof(int)); //
alokacja pamięci dla tablicy prev

    for (int i = 0; i < V; i++) { // pętla po
wierzchołkach

        visited[i] = 0; // ustawienie wartości na 0
(nieodwiedzony)
```

```
        dist[i] = 999; // ustawienie wartości na 999 (nieskończoność)

        prev[i] = -1; // ustawienie wartości na -1 (brak poprzednika)



    }



}

int find_min(int V) { // funkcja znajdująca nieodwiedzony wierzchołek o najmniejszej odległości

    int min = 999; // zmienna przechowująca minimalną odległość

    int min_index = -1; // zmienna przechowująca indeks minimalnego wierzchołka

    for (int i = 0; i < V; i++) { // pętla po wierzchołkach

        if (!visited[i] && dist[i] < min) { // jeśli wierzchołek jest nieodwiedzony i ma mniejszą odległość niż dotychczasowa minimalna

            min = dist[i]; // uaktualnienie minimalnej odległości

            min_index = i; // uaktualnienie indeksu minimalnego wierzchołka



        }



    }

    return min_index; // zwrócenie indeksu minimalnego wierzchołka lub -1, jeśli nie ma takiego



}
```

```
void dijkstra(graph* g, int s, int e) { // funkcja znajdująca najkrótszą ścieżkę w grafie za pomocą algorytmu Dijkstry

    dist[s] = 0; // ustawienie odległości wierzchołka startowego na 0

    int u = find_min(g->V); // znalezienie nieodwiedzonego wierzchołka o najmniejszej odległości

    while (u != -1) { // pętla dopóki taki wierzchołek istnieje

        visited[u] = 1; // oznaczenie wierzchołka jako odwiedzonego

        if (u == e) { // jeśli wierzchołek jest końcowy

            break; // przerwanie pętli



        }

        node* curr = g->adj[u]; // ustawienie wskaźnika na głowę listy sąsiedztwa u

        while (curr != NULL) { // pętla po liście sąsiedztwa u

            int v = curr->x * g->V + curr->y; // obliczenie indeksu wierzchołka sąsiadującego z u

            if (!visited[v] && dist[u] + 1 < dist[v]) { // jeśli wierzchołek jest nieodwiedzony i ma większą odległość niż u plus 1

                dist[v] = dist[u] + 1; // uaktualnienie odległości wierzchołka v

                prev[v] = u; // ustawienie poprzednika wierzchołka v na u



            }

            curr = curr->next; // przejście do następnego węzła w liście
```

```
        }

        u = find_min(g->V); // znalezienie kolejnego
nieodwiedzonego wierzchołka o najmniejszej
odległości



    }



} // zakończ funkcję



void mark_path(char** maze, int width, int height,
int s, int e) { // funkcja zaznaczająca najkrótszą
ścieżkę w labiryncie za pomocą znaku MARK

    int u = e; // ustawienie aktualnego wierzchołka
na końcowy

    while (u != s) { // pętla dopóki aktualny
wierzchołek nie jest startowy

        int x = u / width; // obliczenie
współrzędnej x komórki odpowiadającej wierzchołkowi

        int y = u % width; // obliczenie
współrzędnej y komórki odpowiadającej wierzchołkowi

        if (maze[x][y] != START && maze[x][y] !=
END) { // jeśli komórka nie jest startem ani końcem

            maze[x][y] = MARK; // zastąp znak
komórki znakiem MARK



        }

        u = prev[u]; // przejdź do poprzednika
wierzchołka na najkrótszej ścieżce



    }
```

```
}

int main(){ // główna funkcja programu

    srand(time(NULL)); // ustawienie ziarna
generatora liczb losowych

    int width,height; // zmienne przechowujące
szerokość i wysokość labiryntu

    printf("Podaj liczbę kolumn labiryntu: "); //
wyświetlenie komunikatu

    scanf("%d",&width); // wczytanie szerokości od
użytkownika

    printf("Podaj liczbę wierszy labiryntu: "); //
wyświetlenie komunikatu

    scanf("%d",&height); // wczytanie wysokości od
użytkownika



  // Sprawdź, czy szerokość i wysokość są liczbami
całkowitymi dodatnimi

  char width_str[10], height_str[10];

  sprintf(width_str, "%d", width); // Konwertuj
szerokość na ciąg znaków

  sprintf(height_str, "%d", height); // Konwertuj
wysokość na ciąg znaków

  for (int i = 0; i < strlen(width_str); i++) { //
Sprawdź każdy znak w ciągu szerokości

    if (width_str[i] < '0' || width_str[i] > '9') {
// Jeśli znak nie jest cyfrą

      printf("Wymiary labiryntu muszą być liczbami
całkowitymi dodatnimi.\n");

      return 0;

    }
```

```
  }

  for (int i = 0; i < strlen(height_str); i++) { // Sprawdź każdy znak w ciągu wysokości

    if (height_str[i] < '0' || height_str[i] > '9') { // Jeśli znak nie jest cyfrą

      printf("Wymiary labiryntu muszą być liczbami całkowitymi dodatnimi.\n");

      return 0;

    }

  }

  // Jeśli szerokość lub wysokość nie są liczbami całkowitymi dodatnimi, zwróć 0

  if (width <= 0 || height <= 0) {

    printf("Wymiary labiryntu muszą być liczbami całkowitymi dodatnimi.\n");

    return 0;

  }

  // Jeśli szerokość lub wysokość są liczbami parzystymi, zwróć 0

  if (width % 2 == 0 || height % 2 == 0) {

    printf("Wymiary labiryntu muszą być liczbami nieparzystymi.\n");

    return 0;

  }

  // Jeśli szerokość lub wysokość są mniejsze niż 3, zwróć 0

  if (width < 3 || height < 3) {

    printf("Wymiary labiryntu muszą być większe lub równe 3.\n");

    return 0;
```

```
  }

  // Jeśli szerokość lub wysokość są większe niż 99, zwróć 0

  if (width > 99 || height > 99) {

    printf("Wymiary labiryntu muszą być mniejsze lub równe 99.\n");

    return 0;

  }




    char**maze=malloc(height*sizeof(char*)); // alokacja pamięci dla tablicy labiryntu

    for(int y=0;y<height;y++){ // pętla po wierszach labiryntu

        maze[y]=malloc(width*sizeof(char)); // alokacja pamięci dla wiersza labiryntu

        for(int x=0;x<width;x++){ // pętla po kolumnach labiryntu

            maze[y][x]=WALL; // ustawienie znaku komórki na WALL



        }



    }

    start.x=rand()%(width/2)*2+1; // losowanie współrzędnej x punktu startowego

    start.y=0; // ustawienie współrzędnej y punktu startowego na 0

    end.x=rand()%(width/2)*2+1; // losowanie współrzędnej x punktu końcowego
```

```
    end.y=height-1; // ustawienie współrzędnej y
punktu końcowego na ostatni wiersz

    generate_maze(maze,start,width,height); //
generowanie labiryntu za pomocą algorytmu DFS

    maze[start.y][start.x]=START; // ustawienie
znaku komórki startowej na START

    maze[end.y][end.x]=END; // ustawienie znaku
komórki końcowej na END

    printf("Oto wygenerowany labirynt:\n"); //
wyświetlenie komunikatu

    print_maze(maze,width,height); // wyświetlenie
labiryntu na ekranie

    graph* g = generate_graph(maze, width, height);
// tworzenie grafu na podstawie labiryntu

    init_arrays(width * height); // inicjalizacja
tablic pomocniczych

    int s = start.y * width + start.x; // obliczenie
indeksu wierzchołka startowego

    int e = end.y * width + end.x; // obliczenie
indeksu wierzchołka końcowego

    dijkstra(g, s, e); // znajdowanie najkrótszej
ścieżki w grafie za pomocą algorytmu Dijkstry

    if (dist[e] != 999) { // jeśli istnieje
rozwiązanie

        printf("\nLabirynt ma rozwiązanie.\n\n"); //
wyświetlenie komunikatu

        printf("Najkrótsza ścieżka ma długość
%d.\n\n", dist[e]+1); // wyświetlenie długości
najkrótszej ścieżki

        mark_path(maze, width, height, s,e); //
zaznaczenie najkrótszej ścieżki w labiryncie

        printf("Oto labirynt z zaznaczoną najkrótszą
ścieżką:\n"); // wyświetlenie komunikatu

        print_maze(maze, width, height); //
wyświetlenie labiryntu z zaznaczoną ścieżką
```

```
    }

    else { // jeśli nie istnieje rozwiązanie

        printf("\nLabirynt nie ma rozwiązania.\n");
// wyświetlenie komunikatu



    }

    free_maze(maze,height); // zwolnienie pamięci
labiryntu

    free(visited); // zwolnienie pamięci tablicy
visited

    free(dist); // zwolnienie pamięci tablicy dist

    free(prev); // zwolnienie pamięci tablicy prev

    free(g->adj); // zwolnienie pamięci tablicy list
sąsiedztwa

    free(g); // zwolnienie pamięci grafu

    return 0; // zakończenie programu



} // zakończenie funkcji
```

Sposób wywołania programu

Użytkownik nie może podać jako wymiarów labiryntu danych, które:

- Nie są liczbami całkowitymi dodatnimi, czyli mniejszymi lub równymi zero. Takie dane są nieprawidłowe, ponieważ labirynt musi mieć co najmniej jeden wiersz i jedną kolumnę.
- Są liczbami parzystymi, czyli podzielnymi przez dwa. Takie dane są niepożądane, ponieważ labirynt musi mieć nieparzyste wymiary, aby mieć ściany na brzegach i nie mieć luk w środku.
- Są mniejsze niż trzy, czyli minimalną szerokością i wysokością labiryntu. Takie dane są niepożądane, ponieważ labirynt musi mieć co najmniej jedną ścieżkę i jedno rozwiązanie.
- Są większe niż 99, czyli maksymalną szerokością i wysokością labiryntu. Takie dane są niepożądane, ponieważ labirynt nie może być zbyt duży i trudny do wygenerowania i rozwiązania.

Sposób wywoływania programu jest następujący:

1.Program należy skompilować za pomocą kompilatora języka C, na przykład cc.

Przykładowa komenda to cc -Wall -ansi -pedantic mini_projekt_3.c . Opcje -Wall -

ansi -pedantic służą do włączenia ostrzeżeń i zgodności ze standardem ANSI C.

2.Program należy uruchomić z konsoli.

Przykładowa komenda to ./a.out .

3.Program wypisze na ekranie prośbę o podanie liczby kolumn, a następnie liczby wierszy labiryntu.

4. Jeśli wymiary labiryntu będą zgodne z kryteriami, program wypisze labirynt, gdzie znak S oznacza początek, a znak E oznacza koniec labiryntu.

5. Program określi czy wygenerowany labirynt ma rozwiązanie, jeśli posiada to wypisze długość najkrótszej ścieżki rozwiązującej labirynt oraz wygeneruje ten sam labirynt w którym ta ścieżka będzie oznaczona znakiem * .

Oto przykładowe wywołania programu:

1.

```
Podaj liczbę kolumn labiryntu: 7
Podaj liczbę wierszy labiryntu: 7
Oto wygenerowany labirynt:
# # # # # S #
# . . . # . #
# . # . # . #
# . . # . . #
# . # . . # #
# . . . # # #
# # # E # # #

Labirynt ma rozwiązanie.

Najkrótsza ścieżka ma długość 9.

Oto labirynt z zaznaczoną najkrótszą ścieżką:
# # # # # S #
# . . . # * #
# . # . # * #
# . . # * * #
# . # * * # #
# . . * # # #
# # # E # # #
```

2.

Podaj liczbę kolumn labiryntu: -15

Podaj liczbę wierszy labiryntu: -15

Wymiary labiryntu muszą być liczbami całkowitymi dodatnimi.

3.

Podaj liczbę kolumn labiryntu: 1

Podaj liczbę wierszy labiryntu: 1

Wymiary labiryntu muszą być większe lub równe 3.

4.

Podaj liczbę kolumn labiryntu: 101

Podaj liczbę wierszy labiryntu: 101

Wymiary labiryntu muszą być mniejsze lub równe 99.

5.

Podaj liczbę kolumn labiryntu: 6

Podaj liczbę wierszy labiryntu: 6

Wymiary labiryntu muszą być liczbami nieparzystymi.

Wnioski i spostrzeżenia

- Podczas pisania tego kodu nauczyłem się, jak generować i rozwiązywać labirynty za pomocą algorytmów przeszukiwania w głąb (DFS) i przeszukiwania wszerz (BFS). Zrozumiałem, jak te algorytmy działają i jakie są ich zalety i wady.
- Podczas pisania tego kodu zastosowałem różne struktury danych, takie jak tablice, struktury, stosy i kolejki, do przechowywania i manipulowania danymi. Zrozumiałem, jak te struktury danych są użyteczne i jakie są ich złożoności czasowe i pamięciowe.
- Podczas pisania tego kodu użyłem różnych funkcji, takich jak malloc, free, rand, scanf i printf, do alokowania pamięci, zwalniania pamięci, generowania liczb losowych, pobierania danych od użytkownika i wyświetlania danych na ekranie. Zrozumiałem, jak te funkcje działają i jakie są ich ograniczenia i ryzyka.
- Podczas pisania tego kodu stosowałem różne instrukcje sterujące, takie jak if, else, for, while i return, do wykonywania warunkowej i iteracyjnej logiki programu. Zrozumiałem, jak te instrukcje sterujące wpływają na przepływ programu i jakie są ich konwencje i dobre praktyki.
- Podczas pisania tego kodu używałem różnych zmiennych, takich jak width, height, start, end, maze, solution i parent, do przechowywania wartości i stanu programu. Zrozumiałem, jak te zmienne są inicjalizowane, przypisywane i używane w programie i jakie są ich zakresy i typy.
- Podczas pisania tego kodu napotkałem kilka błędów i ostrzeżeń, takich jak brakujące nawiasy klamrowe, niezgodność typów lub niepoprawne nazwy zmiennych. Udało mi się jednak naprawić wszystkie dostrzeżone przeze mnie błędy.
- Sprawdziłem, czy mój kod działa poprawnie i spełnia wymagania zadania.