Spis treści

Cel projektu

Wyznacz najkrótszą drogę w labiryncie z punktu Start do punktu Stop.

Krawędzie istnieją jedynie pomiędzy sąsiadującymi ze sobą wierzchołkami. Wierzchołki sąsiadują ze sobą jedynie w poziomie i w pionie.

Wagi generowane są w sposób losowy w zakresie od 0 do 10 jako liczby zmiennoprzecinkowe. Przejście pomiędzy komórkami labiryntu obarczone jest wagą równą średniej arytmetycznej dwóch sąsiadujących komórek.

Rozwiązanie problemu

Labirynt został zinterpretowany jako graf kierunkowy z wagami, reprezentowanym przez macierz sąsiedztwa wymiaru $n \times n$, gdzie n – wielkość boku labiryntu. Wartości w macierzy są równe:

- 0, jeżeli nie istnieje przejście z i-tego węzła do j-tego węzła LUB jest to połączenie węzła z samym sobą,
- Wadze przejścia z i-tego węzła do j-tego węzła

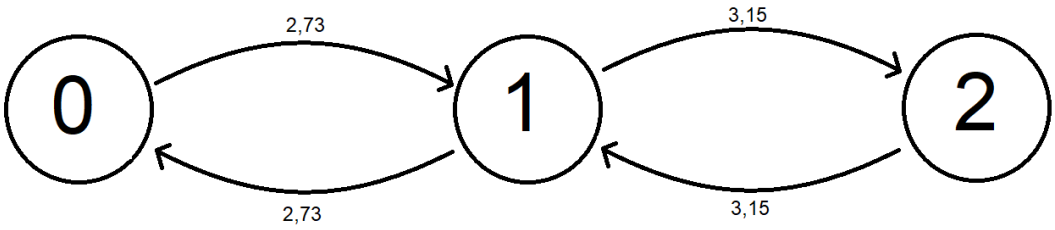
Macierz może przykładowo wyglądać:

$$\begin{bmatrix} 0 & 2{,}73 & 0 \\ 2{,}73 & 0 & 3{,}15 \\ 0 & 3{,}15 & 0 \end{bmatrix}$$

Oznacza to, że:

- Z węzła 0 można dostać się do węzła 1 (waga 2,73)
- Z węzła 1 można dostać się do węzła 0 (waga 2,73) i węzła 2 (waga 3,15)
- Z węzła 2 można dostać się do węzła 1 (waga 3,15)

Graf wygląda następująco:



Założeniem labiryntu jest to, że może nie istnieć droga od punktu startowego do końcowego, ale żadna komórka labiryntu nie jest otoczona ścianami z każdej strony.

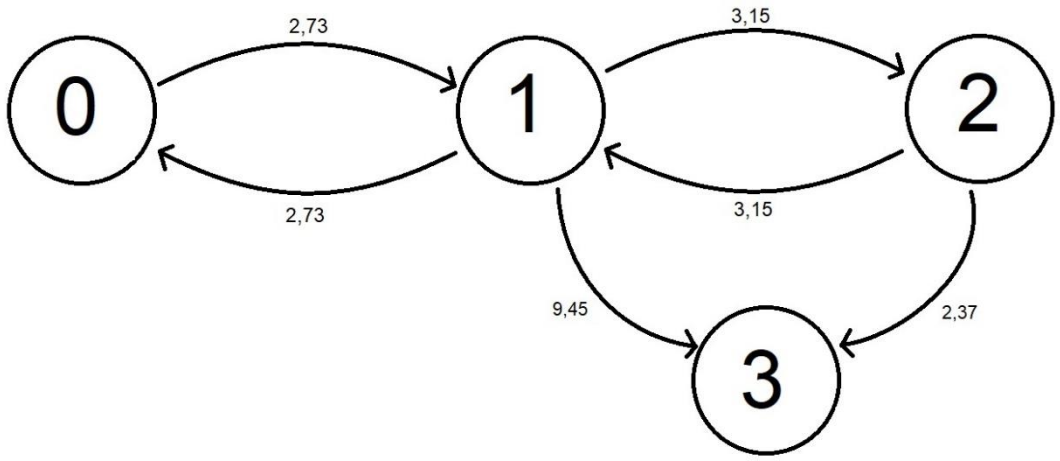
Do przeszukiwania najkrótszej drogi w labiryncie użyłem Algorytmu Dijkstry. Polega on na obliczeniu najkrótszej drogi z węzła początkowego do dowolnego innego węzła.

Zasada działania algorytmu:

1. Oznacz wszystkie węzły jako nieodwiedzone (unvisited).
2. Ustaw odległość wszystkich węzłów jako nieskończoność. Ustaw odległość węzła początkowego A jako 0.
3. Odwiedź nieodwiedzony węzeł z najmniejszą odległością.

a. Oblicz odległości jego sąsiadujących węzłów od węzła A poprzez dodanie dystansu aktualnego węzła i wagi łuku pomiędzy węzłem aktualnym i sąsiadującym.
b. Jeżeli obliczony dystans węzła jest mniejszy niż aktualny, zaktualizuj go oraz oznacz jego poprzednik jako aktualny węzeł
c. Oznacz węzeł jako odwiedzony (visited)
d. Jeżeli wszystkie węzły zostały odwiedzone, zatrzymaj działanie algorytmu
e. Powtórz punkt 3

Przykładowe działanie algorytmu przedstawię na grafie (zakładamy, że węzłem początkowym jest 0):



1. Oznacz wszystkie węzły jako nieodwiedzone.

Odwiedzony?	0	1	2	3
	Nie	Nie	Nie	Nie

2. Ustaw odległość wszystkich węzłów jako nieskończoność. Ustaw odległość węzła początkowego **A** jako 0. Ustaw poprzednika węzła **A** jako -1.

Dystans od punktu początkowego	0	1	2	3
	0	∞	∞	∞

Węzeł	0	1	2	3
Poprzednik	-1			

3. Odwiedź nieodwiedzony węzeł z najmniejszą odległością.
   Odwiedzamy węzeł **0**.

   a. Oblicz odległości jego sąsiadujących węzłów od węzła **A** poprzez dodanie dystansu aktualnego węzła i wagi łuku pomiędzy węzłem aktualnym i sąsiadującym.
   Sąsiadem węzła **0** jest węzeł **1**. Jego odległość od **A** wynosi 0+2,73=2,73

   b. Jeżeli obliczony dystans węzła jest mniejszy niż aktualny, zaktualizuj go oraz oznacz jego poprzednik jako aktualny węzeł
   Dystans węzła **1** od **A** wynosi 2,73 < ∞

Dystans od punktu początkowego	0	1	2	3
	0	2,73	∞	∞

Węzeł	0	1	2	3
Poprzednik	-1	0		

c. Oznacz węzeł jako odwiedzony (visited)

Odwiedzony?	0	1	2	3
	Tak	Nie	Nie	Nie

d. Jeżeli wszystkie węzły zostały odwiedzone, zatrzymaj działanie algorytmu
e. Powtórz punkt 3

3. Odwiedź nieodwiedzony węzeł z najmniejszą odległością.
Odwiedzamy węzeł **1**.

a. Oblicz odległości jego sąsiadujących węzłów od węzła **A** poprzez dodanie dystansu aktualnego węzła i wagi łuku pomiędzy węzłem aktualnym i sąsiadującym.
Sąsiadami węzła **1** jest **0**, **2**, i **3**.
Odległość od **A** do **0** wynosi 0
Odległość od **A** do **2** wynosi 2,73+3,15=5,98
Odległość od **A** do **3** wynosi 2,73+9,45=12,18

b. Jeżeli obliczony dystans węzła jest mniejszy niż aktualny, zaktualizuj go oraz oznacz jego poprzednik jako aktualny węzeł
Dystans **0**: 0 = 0 brak akcji
Dystans **2**: 5,98 < ∞
Dystans **3**: 12,18 < ∞

Dystans od punktu początkowego	0	1	2	3
	0	2,73	5,98	12,18

Węzeł	0	1	2	3
Poprzednik	-1	0	1	1

c. Oznacz węzeł jako odwiedzony (visited)

Odwiedzony?	0	1	2	3
	Tak	Tak	Nie	Nie

d. Jeżeli wszystkie węzły zostały odwiedzone, zatrzymaj działanie algorytmu
e. Powtórz punkt 3

3. Odwiedź nieodwiedzony węzeł z najmniejszą odległością.
Odwiedzamy węzeł **2**

a. Oblicz odległości jego sąsiadujących węzłów od węzła **A** poprzez dodanie dystansu aktualnego węzła i wagi łuku pomiędzy węzłem aktualnym i sąsiadującym.

Sąsiadami węzła **2** jest **1** i **3**.
Odległość od **A** do **1** wynosi 5,98+2,73=8,71
Odległość od **A** do **3** wynosi 5,98+2,37=8,35

b. Jeżeli obliczony dystans węzła jest mniejszy niż aktualny, zaktualizuj go oraz oznacz jego poprzednik jako aktualny węzeł.
Dystans **1**: 8,71 > 2,73 brak akcji
Dystans **3**: 8,35 < 12,18

Dystans od punktu początkowego	0	1	2	3
	0	2,73	5,98	8,35

Węzeł	0	1	2	3
Poprzednik	-1	0	1	2

c. Oznacz węzeł jako oznaczony (visited).

Odwiedzony?	0	1	2	3
	Tak	Tak	Tak	Nie

d. Jeżeli wszystkie węzły zostały odwiedzone, zatrzymaj działanie algorytmu
e. Powtórz punkt 3.

3. Odwiedź nieodwiedzony węzeł z najmniejszą odległością.
Odwiedzamy węzeł **3**.
   a. Oblicz odległości jego sąsiadujących węzłów od węzła **A** poprzez dodanie dystansu aktualnego węzła i wagi łuku pomiędzy węzłem aktualnym i sąsiadującym.
   Sąsiadami węzła **3** jest **1** i **2**.
   Odległość od **A** do **1** wynosi 8,35+2,73=11,08
   Odległość od **A** do **2** wynosi 8,35 +2,37=10,72

   b. Jeżeli obliczony dystans węzła jest mniejszy niż aktualny, zaktualizuj go oraz oznacz jego poprzednik jako aktualny węzeł.
   Dystans **1**: 11,08 > 2,73 brak akcji
   Dystans **2**: 10,72 > 5,98 brak akcji

   c. Oznacz węzeł jako oznaczony (visited).

Odwiedzony?	0	1	2	3
	Tak	Tak	Tak	Tak

   d. Jeżeli wszystkie węzły zostały odwiedzone, zatrzymaj działanie algorytmu.
   KONIEC

Na podstawie zapisanych dystansów wiemy, że najkrótsza odległość od węzła **0** do węzła **3** wynosi 8,35. Na podstawie poprzedników wiemy, że droga (idąc wstecz od węzła **3**) wygląda **3**->**2**->**1**->**0**, czyli prawidłowa droga wygląda **0**->**1**->**2**->**3** .

Szczegóły implementacyjne

Program został napisany w języku C.

Wykorzystano nagłówki biblioteki standardowej:

- stdio.h
- stdlib.h
- time.h

Program składa się z 7 plików.

Plik maze.h zawiera definicję struktury maze_t oraz prototypy funkcji odpowiadających za generację labiryntu.

```c
#ifndef MAZE_H
#define MAZE_H
#include <stdlib.h>

#define WALL -1

typedef struct{
    double **cells;  //cells of the maze
    int start;       //starting cell
    int finish;      //end cell
    size_t size;     //size of one side of the maze
} maze_t;

maze_t* initMaze(int n);
void printMaze(maze_t *maze, int n);
void freeMaze(maze_t *maze);

#endif
```

Plik maze.c zawiera definicje funkcji.

Funkcja initMaze służy alokacji pamięci dla labiryntu oraz wypełnia go losowymi liczbami. Funkcja alokuje potrzebną pamięć, jeżeli się to nie uda to zwraca NULL (jako błąd):

```c
/*Function returns fully-initialized maze*/
maze_t* initMaze(int n){
    int i, j;

    maze_t *newMaze = malloc(n * sizeof *newMaze);
    if(newMaze == NULL)         //error - no memory
        return NULL;

    newMaze->cells = malloc(n * sizeof *newMaze->cells);
    if(newMaze->cells == NULL){ //error - no memory
        free(newMaze);
        return NULL;
    }
```

Wybierane są losowe indeksy komórek startowych i końcowych. Komórka startowa zawsze będzie znajdować się w najwyższym wierszu labiryntu, a końcowa w najniższym. Inicjowana jest też wielkość labiryntu.

```c
//randomize start and end
newMaze->start = rand()%n;
newMaze->finish = rand()%n;
newMaze->size = n;
```

Funkcja używa pętli for w celu alokacji pamięci dla wierszy labiryntu. Jeżeli się to nie uda, to zwalnia pamięć i zwraca NULL. Następnie każdej komórce labiryntu zostaje nadana wartość z przedziału 0,01-9,99 poprzez równanie (rand()%999+1)/100. Dodatkowo, jeżeli spełni się równanie rand()%3==0 (szansa na to jest równa 1/3) to wartość komórki zostaje zmieniona na -1. Program będzie interpretować komórki z wartością -1 jako ściany.

```
//generate random values of cells
for(i = 0; i < n; i++){
    newMaze->cells[i] = malloc(n * sizeof *newMaze->cells[0]);
    if (newMaze->cells[i] == NULL){     //error - no memory
            for(j = 0; j < i; j++)
                free(newMaze->cells[i]);
            free(newMaze->cells);
            free(newMaze);
            return NULL;
    }
    for(j = 0; j < n; j++){
        newMaze->cells[i][j] = (double)(rand() % 999 + 1)/100;
        if(rand() % 3 == 0)     //33,33% chance to change a cell into a wall
            newMaze->cells[i][j] = WALL;
    }
}
```

Ponieważ komórkom startowej i końcowej może zostać przypisana ściana, należy wylosować im nowe wartości z odpowiedniego przedziału.

```
//ensure that start and end cells are not walls
newMaze->cells[0][newMaze->start] = (double)(rand() % 999 + 1)/100;
newMaze->cells[newMaze->size-1][newMaze->finish] = (double)(rand() % 999 + 1)/100;
```

Następnie wywoływana jest funkcja ifSurrounded oraz funkcja zwraca utworzoną strukturę labiryntu.

```
    ifSurrounded(newMaze);
    return newMaze;
}
```

Funkcja ifSurrounded sprawdza, czy którakolwiek komórka labiryntu jest otoczona ścianami ze wszystkich stron. Jeżeli tak się dzieje, to losowa ściana zostaje zamieniona w normalną komórkę. Jako pierwszymi funkcja zajmuje się rogami labiryntu, czyli:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3

(Jest to przykładowy labirynt 4x4 z indeksami wewnątrz komórek)

```
/*  *Function checks if one cell of the maze is surrounded
    *by walls from every side */
static void ifSurrounded (maze_t *maze){
    int i, j;

    //check if corners of the maze are closed (surrounded)
    if(maze->cells[0][0] != WALL && maze->cells[0][1] == WALL && maze->cells[1][0] == WALL){  //left-upper corner
        int direction = rand() % 2;
        switch (direction){
        case 0: //change right wall
            maze->cells[0][1] = (double)(rand() % 999 + 1)/100;
            break;

        case 1: //change bottom wall
            maze->cells[1][0] = (double)(rand() % 999 + 1)/100;
            break;
        }
    }
    if(maze->cells[maze->size - 1][0] != WALL && maze->cells[maze->size - 1][1] == WALL && maze->cells[maze->size - 2][0] == WALL){//
    left-bottom corner
        int direction = rand() % 2;
        switch (direction){
        case 0: //change upper wall
            maze->cells[maze->size - 2][0] = (double)(rand() % 999 + 1)/100;
            break;
        case 1: //change right wall
            maze->cells[maze->size - 1][1] = (double)(rand() % 999 + 1)/100;
            break;
        }
    }

    if(maze->cells[0][maze->size - 1] != WALL && maze->cells[0][maze->size - 2] == WALL && maze->cells[1][maze->size - 1] == WALL){ //
    right-upper corner
        int direction = rand() % 2;
        switch (direction){
        case 0: //change left wall
            maze->cells[0][maze->size - 2] = (double)(rand() % 999 + 1)/100;
            break;
        case 1: //change bottom wall
            maze->cells[1][maze->size - 1] = (double)(rand() % 999 + 1)/100;
            break;
        }
    }

    if(maze->cells[maze->size - 1][maze->size - 1] != WALL && maze->cells[maze->size - 1][maze->size - 2] == WALL && maze->cells
    [maze->size - 2][maze->size - 1] == WALL){ //right-bottom corner
        int direction = rand() % 2;
        switch (direction){
        case 0: //change left wall
            maze->cells[maze->size - 1][maze->size - 2] = (double)(rand() % 999 + 1)/100;
            break;
        case 1: //change bottom wall
            maze->cells[maze->size - 2][maze->size - 1] = (double)(rand() % 999 + 1)/100;
            break;
        }
    }

```

Następnie funkcja sprawdza komórki leżące na zewnętrznych ścianach labiryntu (niebędących rogami):

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3

```
    //check sides of the maze
    for(i = 1; i < maze->size - 1; i++){
        if(maze->cells[0][i] != WALL && maze->cells[0][i-1] == WALL && maze->cells[0][i+1] == WALL && maze->cells[1][i] == WALL){ //upper row
            int direction = rand() % 3;
            switch (direction){
            case 0: //change left
                maze->cells[0][i-1] = (double)(rand() % 999 + 1)/100;
                break;
            case 1: //change bottom
                maze->cells[1][i] = (double)(rand() % 999 + 1)/100;
                break;
            case 2: //change right
                maze->cells[0][i+1] = (double)(rand() % 999 + 1)/100;
                break;
            }
        }
        if(maze->cells[maze->size - 1][i] != WALL && maze->cells[maze->size - 1][i-1] == WALL && maze->cells[maze->size - 1][i+1] == WALL && maze->cells[maze->size - 2][i] == WALL){ //bottom row
            int direction = rand() % 3;
            switch (direction){
            case 0: //change left
                maze->cells[maze->size - 1][i-1] = (double)(rand() % 999 + 1)/100;
                break;
            case 1: //change upper
                maze->cells[maze->size - 2][i] = (double)(rand() % 999 + 1)/100;
                break;
            case 2: //change right
                maze->cells[maze->size - 1][i+1] = (double)(rand() % 999 + 1)/100;
                break;
            }
        }
        if(maze->cells[i][0] != WALL && maze->cells[i-1][0] == WALL && maze->cells[i][1] == WALL && maze->cells[i+1][0] ){ //left column
            int direction = rand() % 3;
            switch (direction){
            case 0: //change upper
                maze->cells[i-1][0] = (double)(rand() % 999 + 1)/100;
                break;
            case 1: //change right
                maze->cells[i][1] = (double)(rand() % 999 + 1)/100;
                break;
            case 2: //change down
                maze->cells[i+1][0] = (double)(rand() % 999 + 1)/100;
                break;
            }
        }
        if(maze->cells[i][maze->size - 1] != WALL && maze->cells[i-1][maze->size - 1] == WALL && maze->cells[i][maze->size - 2] == WALL && maze->cells[i+1][maze->size - 1] ){ //right column
            int direction = rand() % 3;
            switch (direction){
            case 0: //change upper
                maze->cells[i-1][maze->size - 1] = (double)(rand() % 999 + 1)/100;
                break;
            case 1: //change left
                maze->cells[i][maze->size - 2] = (double)(rand() % 999 + 1)/100;
                break;
            case 2: //change bottom
                maze->cells[i+1][maze->size - 1] = (double)(rand() % 999 + 1)/100;
                break;
            }
        }
    }
```

Na koniec sprawdzane są wewnętrzne komórki labiryntu:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3

```
    //check if inner cells are surrounded
    for(i = 1; i < maze->size - 2; i++)
        for(j = 1; j < maze->size - 2; j++)
            if(maze->cells[i][j] != WALL && maze->cells[i-1][j] == WALL && maze->cells[i+1][j] == WALL && maze->cells[i][j-1] == WALL &&
            maze->cells[i][j+1] == WALL){  //if cell is surrounded
                int direction = rand() % 4;
                switch (direction){
                case 0: //change upper wall
                    maze->cells[i-1][j] = (double)(rand() % 999 + 1)/100;
                    break;
                case 1: //change bottom wall
                    maze->cells[i+1][j] = (double)(rand() % 999 + 1)/100;
                    break;
                case 2: //change left wall
                    maze->cells[i][j-1] = (double)(rand() % 999 + 1)/100;
                    break;
                case 3: //change right wall
                    maze->cells[i][j+1] = (double)(rand() % 999 + 1)/100;
                    break;
                }
            }
}
```

Funkcja printMaze drukuje labirynt na stdout.

```
185   /* Function prints maze to stdout*/
186   void printMaze(maze_t *maze, int n){
187       int i, j;
188
189       printf("\nGenerated maze:\n\n");
190
191       //first wall
192       printf("##");
193       for(i = 0; i < n; i++){
194           if(i == maze->start)
195               printf("      ");
196           else
197               printf("#####");
198       }
199       printf("###");
200       printf("\n");
```

W celu wyświetlania labiryntu funkcja sprawdza wszystkie komórki labiryntu po kolei. Jeżeli komórka ma wartość -1 to oznacza, że jest to ściana i program zamiast wartości tej komórki wyświetla ścianę (czyli „####”)

```
202     //rows of the maze
203     for(i = 0; i < n; i++){
204         printf("## ");
205         for(j = 0; j < n; j++){
206             if(maze->cells[i][j] == WALL)
207                 printf("#### ");
208             else
209                 printf("%.2lf ", maze->cells[i][j]);
210         }
211         printf(" ##\n");
212     }
213     printf("##");
```

```
215     //bottom wall
216     for(i = 0; i < n; i++){
217         if(i == maze->finish)
218             printf("      ");
219         else
220             printf("#####");
221     }
222     printf("###\n\n");
223 }
```

Przykładowe wyświetlanie labiryntu wielkości 3 i 6:

```
Generated maze:

##      ###############
## 0.42 0.70 2.72  ##
## #### 6.54 7.44  ##
## 7.73 3.57 2.31  ##
##      ###############
```

```
Generated maze:

#######      ########################
## 2.71 6.74 4.44 #### #### 7.96  ##
## 3.44 0.08 6.10 0.81 1.98 7.45  ##
## 1.11 #### #### 7.88 6.43 7.35  ##
## 5.11 8.98 #### 6.31 #### ####  ##
## 5.93 7.76 9.99 9.51 7.69 9.73  ##
## 3.97 #### 8.25 8.12 7.38 6.54  ##
##################      ##############
```

Funkcja freeMaze zwalnia pamięć zajmowaną przez strukturę maze_t

```
/* Function frees memory allocated for the maze */
void freeMaze(maze_t *maze){
    int i;
    for(i = 0; i < maze->size; i++)
        free(maze->cells[i]);
    free(maze->cells);
    free(maze);
}
```

Plik matrix.h zawiera definicję struktury matrix_t odzwierciedlającą macierz sąsiedztwa oraz prototypy funkcji operujących na niej. Zmienna adjMatrix jest macierzą sąsiedztwa, a nodeMatrix jest macierzą węzłów, która przypisuje komórkom labiryntu numer węzła.

```c
#define INF 2147483647  //representation of infinity - maximum value stored by int

typedef struct matrix{
    int n;                      //number of nodes
    double **adjMatrix;         //representation of adjacency matrix
    int **nodeMatrix;           //representatnion of maze with numbered nodes
} matrix_t;

matrix_t *initMatrix(maze_t *maze);
void freeMatrix(matrix_t *matrix, int size);

#endif
```

Plik matrix.c zawiera definicje funkcji. Funkcja initMatrix służy do alokacji pamięci potrzebnej macierzom sąsiedztwa i węzłów oraz je uzupełnia. Jeżeli funkcja nie znajdzie pamięci do alokacji to zwraca NULL.

```c
47  /* Funcion returns a fully-initialized matrix */
48  matrix_t *initMatrix(maze_t *maze){
49      int i, j;
50
51      matrix_t *newMatrix = malloc(sizeof *newMatrix);
52      if (newMatrix == NULL)              //error - no memory
53          return NULL;
54
55      newMatrix->n = 0;
56
57      newMatrix->nodeMatrix = malloc(maze->size * sizeof newMatrix->nodeMatrix);
58      if(newMatrix->nodeMatrix == NULL){  //error - no memory
59          free(newMatrix);
60          return NULL;
61      }
62
63      for(i = 0; i < maze->size; i++){
64          newMatrix->nodeMatrix[i] = calloc(maze->size, sizeof *newMatrix->nodeMatrix);
65          if(newMatrix->nodeMatrix[i] == NULL){   //error - no memory
66              for(j = 0; j < i; j++)
67                  free(newMatrix->nodeMatrix[j]);
68              free(newMatrix->nodeMatrix);
69              free(newMatrix);
70              return NULL;
71          }
72      }
73
```

Po rezerwacji pamięci dla macierzy węzłów funkcja uzupełnia ją, licząc przy tym węzły. Jeżeli komórka labiryntu nie jest ścianą, to zostaje jej nadany numer węzła.

```c
74      //count nodes, fill the nodeMatrix
75      for(i = 0; i < maze->size; i++)
76          for(j = 0; j < maze->size; j++)
77              if(maze->cells[i][j] != WALL)
78                  newMatrix->nodeMatrix[i][j] = newMatrix->n++;
79
```

Zostaje zarezerwowane miejsce na macierz sąsiedztwa.

```
81      newMatrix->adjMatrix = malloc(newMatrix->n * sizeof *newMatrix->adjMatrix);
82      if(newMatrix->adjMatrix == NULL){       //error - no memory
83          for(i = 0; i < maze->size; i++)
84              free(newMatrix->nodeMatrix[i]);
85          free(newMatrix->nodeMatrix);
86          free(newMatrix);
87          return NULL;
88      }
89
90      for(i = 0; i < newMatrix->n; i++){
91          newMatrix->adjMatrix[i] = calloc(newMatrix->n, sizeof *newMatrix->adjMatrix[i]);
92          if(newMatrix->adjMatrix[i] == NULL){    //error - no memory
93              for(j = 0; j < i; j++)
94                  free(newMatrix->adjMatrix[j]);
95              free(newMatrix->adjMatrix);
96              for(j = 0; j < maze->size; j++)
97                  free(newMatrix->nodeMatrix[j]);
98              free(newMatrix->nodeMatrix);
99              free(newMatrix);
100             return NULL;
101         }
102     }
```

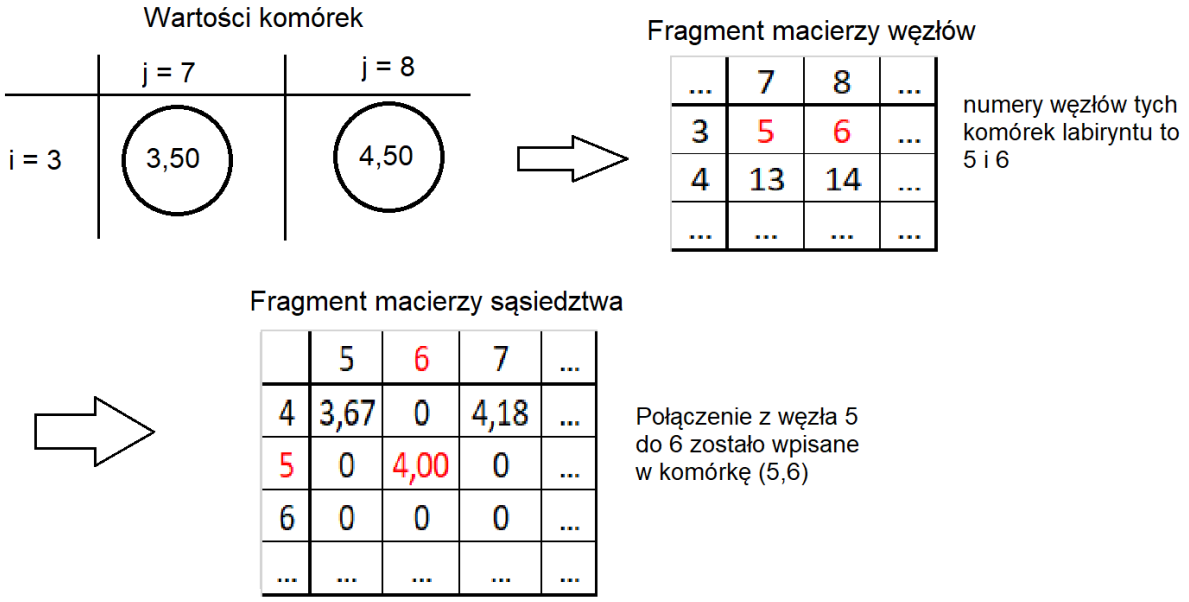
Następnie funkcja uzupełnia macierz sąsiedztwa. Zadane warunki można wytłumaczyć jako:

„Jeżeli komórka [i][j] nie jest ścianą i komórka po lewej ([i][j-1]) / po prawej ([i][j+1]) / na górze ([i-1][j]) / na dole ([i+1][j] też nie jest ścianą, to dodaj połączenie w odpowiedniej komórce macierzy sąsiedztwa”.

Wartość tego połączenia to średnia arytmetyczna wartości komórek labiryntu. Funkcja posługuje się macierzą węzłów, żeby wiedzieć w którą komórkę macierzy sąsiedztwa wpisać połączenie.

```
104     //fill adjacency matrix
105     for(i = 0; i < maze->size; i++)
106         for(j = 0; j < maze->size; j++){
107             if(j != 0 && maze->cells[i][j-1] != WALL && maze->cells[i][j] != WALL){ //if cell to the left is not a wall
108                 //add connection
109                 newMatrix->adjMatrix[newMatrix->nodeMatrix[i][j]][newMatrix->nodeMatrix[i][j-1]] = (maze->cells[i][j-1] + maze->cells[i]
                    [j]) / 2;
110             }
111             if(j != maze->size - 1 && maze->cells[i][j+1] != WALL && maze->cells[i][j] != WALL){ //if cell to the right is not a wall
112                 //add connection
113                 newMatrix->adjMatrix[newMatrix->nodeMatrix[i][j]][newMatrix->nodeMatrix[i][j+1]] = (maze->cells[i][j+1] + maze->cells[i]
                    [j]) / 2;
114             }
115             if(i != maze->size - 1 && maze->cells[i+1][j] != WALL && maze->cells[i][j] != WALL){ //if cell to the bottom is not a wall
116                 //add connection
117                 newMatrix->adjMatrix[newMatrix->nodeMatrix[i][j]][newMatrix->nodeMatrix[i+1][j]] = (maze->cells[i+1][j] + maze->cells[i]
                    [j]) / 2;
118             }
119             if(i != 0 && maze->cells[i-1][j] != WALL && maze->cells[i][j] != WALL){ //if cell up is not a wall
120                 //add connection
121                 newMatrix->adjMatrix[newMatrix->nodeMatrix[i][j]][newMatrix->nodeMatrix[i-1][j]] = (maze->cells[i+-1][j] + maze->cells[i]
                    [j]) / 2;
122             }
123         }
```

Działa to według następującego przykładu (rozpatrujemy komórkę labiryntu o współrzędnych (3,7):



Funkcja wywyołuje printNodeMatrix, która drukuje na stdout wygląd macierzy węzłów analogicznie do printMaze. Ponownie, jeżeli wartość komórki labiryntu wynosi -1, to zostaje wydrukowana ściana.

```c
static void printNodeMatrix(maze_t *maze, matrix_t *matrix){
    int i, j;

    printf("Maze represented with numbered nodes:\n\n");
    //first wall
    printf("##");
    for(i = 0; i < maze->size; i++){
        if(i == maze->start)
            printf("      ");
        else
            printf("#####");
    }
    printf("###\n");

    //rows of the maze
    for(i = 0; i < maze->size; i++){
        printf("## ");
        for(j = 0; j < maze->size; j++){
            if(maze->cells[i][j] == WALL)
                printf("#### ");
            else
                printf("%3d  ", matrix->nodeMatrix[i][j]);
        }
        printf(" ##\n");
    }

    //bottom wall
    printf("##");
    for(i = 0; i < maze->size; i++){
        if(i == maze->finish)
            printf("      ");
        else
            printf("#####");
    }
    printf("###\n\n");

    printf("Number of nodes = %d\n", matrix->n);
    printf("Starting node = %d\n", matrix->nodeMatrix[0][maze->start]);
    printf("End node = %d\n\n", matrix->nodeMatrix[maze->size - 1][maze->finish]);
}
```

Przykładowe wyświetlenie macierzy węzłów:

```
Maze represented with numbered nodes:

############      ###
##   0  ####   1   ##
##   2  ####   3   ##
## ####   4    5   ##
#######      ########

Number of nodes = 6
Starting node = 1
End node = 4
```

Ostatnią czynnością funkcji jest warunkowe wyświetlenie macierzy incydencji. Jest ona wyświetlana tylko, jeżeli podczas kompilacji zdefiniowano zmienną DEBUG. Wyświetlenie macierzy jest pomocne w celu wyszukiwania błędów w programie. Jeżeli wielkość macierzy jest większa niż 15, to użytkownik zostanie zapytany, czy chcę ją wyświetlić.

```
127     #ifdef DEBUG
128     //print adjacency matrix to stdout
129     if(newMatrix->n > 15){  //disclaimer - too big adjacency matrix
130         char c;
131         printf("Warning: size of the adjacency matrix (%d) is too big and may not be displayed properly\n", newMatrix->n);
132         do{
133             printf("Do you want to print it? [y/n]");
134             c = fgetc(stdin);
135             fflush(stdin);
136         } while (c != 'y' && c != 'n');
137 
138         if(c == 'y'){
139             printf("\nAdjacency matrix:\n");
140             for(i = 0; i < newMatrix->n; i++){
141                 for(j = 0; j < newMatrix->n; j++)
142                     printf("%2.2lf ", newMatrix->adjMatrix[i][j]);
143                 printf("\n\n");
144             }
145         }
146     }
147     else{
148         printf("Adjacency matrix:\n");
149             for(i = 0; i < newMatrix->n; i++){
150                 for(j = 0; j < newMatrix->n; j++)
151                     printf("%2.2lf ", newMatrix->adjMatrix[i][j]);
152                 printf("\n\n");
153             }
154     }
155     #endif
156 
157     return newMatrix;
158 }
```

Przykładowe wyświetlenie macierzy sąsiedztwa dla 6 węzłów i zapytanie użytkownika, czy drukować dużą macierz:

```
Adjacency matrix:
0.00 1.99 0.00 5.79 0.00 0.00

1.99 0.00 0.00 0.00 0.00 0.00

0.00 0.00 0.00 9.18 8.93 0.00

5.79 0.00 9.18 0.00 0.00 4.84

0.00 0.00 8.93 0.00 0.00 4.60

0.00 0.00 0.00 4.84 4.60 0.00
```

```
Warning: size of the adjacency matrix (17) is too big and may not be displayed properly
Do you want to print it? [y/n]y
```

Funkcja freeMatrix zwalnia pamięć zarezerwowaną na strukturę macierzy.

```c
/*Function frees memory allocated for the matrix*/
void freeMatrix(matrix_t *matrix, int size){
    int i;

    for(i = 0; i < matrix->n; i++)
        free (matrix->adjMatrix[i]);
    free(matrix->adjMatrix);

    for(i = 0; i < size; i++)
        free(matrix->nodeMatrix[i]);
    free(matrix->nodeMatrix);

    free(matrix);
}
```

Plik dijkstra.h zawiera strukturę listy liniowej oraz prototypy funkcji.

```c
#ifndef DIJKSTRA_H
#define DIJKSTRA_H

#include "maze.h"
#include "matrix.h"

typedef struct order{
    int nodeNumber;
    struct order *next;
} orderList_t;  //structure of linked list

int pushOrder(orderList_t **head, int newVertex);
void printSolution(int endNode,int startNode, double *dist, int *predecessor, int n);
void dijkstra(matrix_t *matrix, maze_t *maze);

#endif
```

Plik dijkstra.c zawiera definicje funkcji.

Funkcja dijkstra jest odwzorowaniem algorytmu Dijkstry. Tablica dist odpowiada dystansom od węzła startowego do i-tego węzła, tablica visited przechowuje informacje o tym, czy i-ty węzeł został odwiedzony, tablica predecessor zapamiętuje poprzednika i-tego węzła. Funkcja alokuje pamięć potrzebną na te tablice, w przypadku błędu zwalnia pamięć i kończy działanie wyświetlając komunikat błędu.

```
76  /* Function that performs Dijkstra's Algorithm */
77  void dijkstra(matrix_t *matrix, maze_t *maze){
78      int i, j;
79
80      double *dist = malloc(matrix->n * sizeof *dist);
81      if(dist == NULL){               //error - not enough memory
82          fprintf(stderr, "Error: not enough memory for Dijkstra's Algorithm\n");
83          return;
84      }
85
86      int *visited = calloc(matrix->n, sizeof visited);   // visited == 1, unvisited == 0
87      if(visited == NULL){            //error - not enough memory
88          free(dist);
89          fprintf(stderr, "Error: not enough memory for Dijkstra's Algorithm\n");
90          return;
91      }
92
93      int *predecessor = malloc(matrix->n * sizeof *predecessor);
94      if(predecessor == NULL){        //error - not enough memory
95          free(visited);
96          free(dist);
97          fprintf(stderr, "Error: not enough memory for Dijkstra's Algorithm\n");
98          return;
99      }
100
101     int startNode = matrix->nodeMatrix[0][maze->start];
102     int endNode = matrix->nodeMatrix[maze->size - 1][maze->finish];
```

Funkcja ustawia dystans wszystkich wierzchołków jako nieskończoność, a wierzchołkowi startowemu 0. Dodatkowo, poprzednikiem wierzchołka startowego umownie jest wierzchołek -1.

```
104     for(i = 0; i < matrix->n; i++)
105         dist[i] = INF;
106
107     dist[startNode] = 0;
108     predecessor[startNode] = -1;
```

Następnie funkcja realizuje algorytm. Z każdym powtórzeniem pętli zmienna currNode przyjmuje wartość nieodwiedzonego węzła z najmniejszą odległością od węzła startowego (za pomocą funkcji minDist). Następnie sprawdzany jest warunek:

„Jeżeli istnieje połączenie z aktualnego węzła do j-tego węzła (czyli są sąsiadami), ten węzeł jest nieodwiedzony, dystans aktualnego węzła jest znany i nowy dystans j-tego węzła jest mniejszy niż jego aktualny, to:

1. *Zaktualizuj dystans*
2. *Zaktualizuj poprzednika j-tego węzła”*

Dodatkowo, do algorytmu został dodany nowy warunek usprawniający jego działanie: jeżeli odwiedzonym węzłem jest węzeł końcowy, to zatrzymaj działanie algorytmu. Działa to, ponieważ algorytm w pierwszej kolejności sprawdza węzły o najkrótszej drodze, czyli w zasadzie najkrótszą drogę aktualnie. Pierwsza taka droga, która napotka na węzeł końcowy, będzie jednocześnie najkrótszą możliwą drogą.

Na koniec wyświetlane jest rozwiązanie labiryntu funkcją printSolution oraz zwolniona zostaje pamięć.

```
110     for(i = 0; i < matrix->n; i++){
111         int currNode = minDist(dist, visited, matrix->n);
112         visited[currNode] = 1;
113 
114         for(j = 0; j < matrix->n; j++)
115             if(visited[j] != 1 && matrix->adjMatrix[currNode][j] && dist[currNode] != INF &&dist[currNode] + matrix->adjMatrix[currNode]
            [j] < dist[j]){
116                 dist[j] = dist[currNode] + matrix->adjMatrix[currNode][j];
117                 predecessor[j] = currNode;
118                 if(j == endNode)    //if reached end node
119                     goto END;       //stop processing algorithm
120             }
121 
122     }
123     END:
124     printSolution(endNode, startNode, dist, predecessor, matrix->n);
125     free(visited);
126     free(dist);
127     free(predecessor);
128 }
```

Funkcja minDist przechodzi po wszystkich nieodwiedzonych wierzchołkach i zwraca numer tego wierzchołka, który ma najmniejszą długość od wierzchołka startowego.

```
51  static double minDist(double *dist, int *visited, int n){
52      double minDist = INF;
53      int minNode;
54      int i;
55      for(i = 0; i < n; i++)
56          if(visited[i] == 0 && dist[i] <= minDist){  //look for unvisited node with smallest distance
57              minDist = dist[i];
58              minNode = i;
59          }
60 
61      return minNode;
62  }
```

Funkcja printSolution służy wypisywaniu długości drogi z punktu startowego do końcowego oraz pokazuje tę drogę. Jeżeli droga nie istnieje, to pojawia się odpowiedni komunikat.

```
5   void printSolution(int endNode,int startNode, double *dist, int *predecessor, int n){
6   
7       if(dist[endNode] == INF){   //no route
8           printf("There is no route between starting node and end node.\n");
9           return;
10      }
```

Kolejność węzłów będących najkrótszą drogą jest reprezentowana przez listę liniową. Szukanie drogi zaczynamy od końca i dzięki tablicy poprzedników (predecessor) możemy przechodzić przez poprzedników aż do punktu startowego, którego poprzednikiem jest ustalone -1. Dodatkowo, za każdym razem kiedy tworzony jest nowy element listy sprawdzane jest, czy została zarezerwowana pamięć i jeżeli nie, to cała lista jest usuwana.

```
18      //initialization of linked list
19      order->nodeNumber = endNode;
20      order->next = NULL;
21      int next = predecessor[endNode];
22
23      do{
24          if(pushOrder(&order, next)){    //add element to the beginning of linked list
25              //if pushOrder return 1 - error, not enough memory
26              fprintf(stderr, "Error: not enough memory for Dijkstra's Algorithm\n");
27              while(order != NULL){       //free memory allocated for linked list
28                  orderList_t *temp = order;
29                  order = order->next;
30                  free(temp);
31              }
32              return;
33          }
34          next = predecessor[next];       //proceed to the next node
35      } while ((predecessor[next] != -1));
```

Funkcja pushOrder dodaje nowy element na początek listy, popychając istniejące już elementy.

```
66  int pushOrder(orderList_t **head, int newVertex){
67      orderList_t *newNode = malloc(sizeof *newNode);
68      if(newNode == NULL)     //error - no enough memory
69          return 1;
70      newNode->nodeNumber = newVertex;
71      newNode->next = (*head);
72      (*head) = newNode;
73      return 0;
74  }
```

Na koniec funkcja printSolution wypisuje najkrótszą drogę, zwalniając jednocześnie pamięć zarezerwowaną przez listę liniową.

```
37      //print results
38      printf("The shortest distance from start to end is: %g\n",dist[endNode]);
39      printf("%d", startNode);
40      while(order != NULL){       //print elements & free linked list
41          orderList_t *temp = order;
42          printf(" -> %d", order->nodeNumber);
43          order = order->next;
44          free(temp);
45      }
46      printf("\n");
47  }
```

Przykładowe wydrukowanie wyników (brak drogi i istniejąca droga):

```
Maze represented with numbered nodes:

############      ########
##   0    1    2  ####  ##
## #### #### #### ####  ##
##   3    4    5  ####  ##
##   6    7    8    9   ##
############      ########

Number of nodes = 10
Starting node = 2
End node = 8

There is no route between starting node and end node.
```

```
Maze represented with numbered nodes:

############      ########
##   0    1    2    3   ##
## #### ####   4  ####  ##
##   5    6    7  ####  ##
##   8    9   10   11   ##
##      ##################

Number of nodes = 12
Starting node = 2
End node = 8

The shortest distance from start to end is: 30.94
2 -> 4 -> 7 -> 6 -> 5 -> 8
```

Plik main.c wywołuje wszystkie pozostałe funkcje w celu utworzenia oraz rozwiązania labiryntu. Żeby labirynt był losowy użyta została funkcja srand(time(0)). Jeżeli nie podano argumentu lub rozmiar labiryntu jest błędny (prawidłowa wielkość to 3-15), to zostaje wyświetlony komunikat błędu.

```c
int main(int argc, char** argv){
    srand(time(0));

    if(argc == 1){      //error - no arguments
        fprintf(stderr, "Error: too few arguments\n");
        fprintf(stderr, "Usage:\n");
        fprintf(stderr, "<filename> <size_of_maze>\n");
        fprintf(stderr, "Size: 3-15\n");
        return 1;
    }

    int size = atoi(argv[1]);
    if(size < 3 || size > 15){       //error - wrong size
        fprintf(stderr, "Error: wrong maze size\n");
        fprintf(stderr, "Size: 3-15\n");
        return 2;
    }
```

Zostaje zainicjowany labirynt funkcją initMaze oraz jest wyświetlany funkcją printMaze.

```c
    maze_t *maze = initMaze(size);
    if(maze == NULL){       //error - no allocated memory for maze
        fprintf(stderr, "Error: not enough memory for creating a maze\n");
        return 3;
    }
    printMaze(maze, size);
```

Zostaje zainicjowana struktura przechowująca macierze węzłów i sąsiedztwa funkcją initMatrix. W przypadku błędu pamięć zostaje zwolniona.

```c
    matrix_t *matrix = initMatrix(maze);
    if(matrix == NULL){     //error - no allocated memory for matrix
        fprintf(stderr, "ERROR: not enough memory for creating an adjacency matrix\n");
        freeMaze(maze);
        return 4;
    }
```

Na koniec zostaje wywołana funkcja Dijkstra oraz zwolniona zostaje zarezerwowana pamięć.

```
40      dijkstra(matrix, maze);
41
42      freeMaze(maze);
43      freeMatrix(matrix, size);
44      return 0;
45
46  }
```

Kompilację programu można przeprowadzić na 2 sposoby:

gcc main.c maze.c matrix.c dijkstra.c (-o [*nazwa*])

gcc main.c maze.c matrix.c dijkstra.c (-o [*nazwa*]) -DDEBUG
- *opcja z wyświetleniem macierzy incydencji*

Możliwe kody błędu programu:

1 - Brak argumentu wywołania

2 - Błędny rozmiar labiryntu

3 - Brak pamięci dla labiryntu

4 - Brak pamięci dla struktury macierzy

Sposób wywołania programu

Program wywołujemy w terminalu, jako argument podajemy wielkość labiryntu. Przykładowe wywołania:

```
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)> .\a.exe 4

Generated maze:

############      ########
## #### #### 1.87 ####  ##
## #### 9.76 8.76 9.76  ##
## 2.31 #### 5.98 4.82  ##
## 4.11 2.94 8.61 6.51  ##
############      ########

Maze represented with numbered nodes:

############      ########
## #### ####   0  ####  ##
## ####   1    2    3   ##
##   4  ####   5    6   ##
##   7    8    9   10   ##
############      ########

Number of nodes = 11
Starting node = 0
End node = 9

The shortest distance from start to end is: 19.98
0 -> 2 -> 5 -> 9
```

```
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)> .\a.exe 7

Generated maze:

################################      ###
## #### 4.99 8.97 0.16 #### 8.37 9.85  ##
## 2.78 9.35 #### 6.01 3.33 #### ####  ##
## #### #### 1.99 9.06 #### 8.03 6.44  ##
## 0.77 #### 0.61 4.03 0.85 #### 4.77  ##
## 5.15 #### #### 6.77 3.24 2.41 9.54  ##
## #### 6.82 8.66 6.86 8.71 #### 4.62  ##
## 4.00 9.12 9.39 0.61 7.92 4.46 8.18  ##
############################      ########

Maze represented with numbered nodes:

################################      ###
## ####   0    1    2  ####   3    4   ##
##   5    6  ####   7    8  #### ####  ##
## #### ####   9   10  ####  11   12   ##
##  13  ####  14   15   16  ####  17   ##
##  18  #### #### 19   20   21   22   ##
## ####  23   24   25   26  ####  27   ##
##  28   29   30   31   32   33   34   ##
############################      ########

Number of nodes = 35
Starting node = 4
End node = 33

There is no route between starting node and end node.
```

```
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe a
Error: wrong maze size
Size: 3-15
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe a2
Error: wrong maze size
Size: 3-15
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe 2a
Error: wrong maze size
Size: 3-15
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe a3
Error: wrong maze size
Size: 3-15
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe 3a

Generated maze:

##      #############
## 7.81 5.49 7.67  ##
## 2.97 9.24 9.08  ##
## #### 1.48 1.94  ##
#######      ########

Maze represented with numbered nodes:

##      #############
##   0    1    2   ##
##   3    4    5   ##
## ####   6    7   ##
#######      ########

Number of nodes = 8
Starting node = 0
End node = 6

The shortest distance from start to end is: 16.855
0 -> 3 -> 4 -> 6
```

```
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe 3a\ 5

Generated maze:

##      #############
## 4.48 #### ####  ##
## 2.38 1.19 2.80  ##
## 4.71 #### 2.69  ##
##      #############

Maze represented with numbered nodes:

##      #############
##   0  #### ####  ##
##   1    2    3   ##
##   4  ####   5   ##
##      #############

Number of nodes = 6
Starting node = 0
End node = 4

The shortest distance from start to end is: 6.975
0 -> 1 -> 4
```

```
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe 26
Error: wrong maze size
Size: 3-15
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)\maze> .\a.exe 16
Error: wrong maze size
Size: 3-15
```

```
PS C:\Users\mikip\OneDrive\Dokumenty\Notatki ze szkoły\Semestr 1\Lab. Podstawy Informatyki\Miniprojekt 3 (maze)> .\a.exe 3

Generated maze:

##      #############
## 1.39 5.53 ####  ##
## #### 9.11 2.36  ##
## 4.48 6.02 1.90  ##
#############      ###

Maze represented with numbered nodes:

##      #############
##   0    1  ####  ##
## ####   2    3   ##
##   4    5    6   ##
#############      ###

Number of nodes = 7
Starting node = 0
End node = 6

Adjacency matrix:
0.00 3.46 0.00 0.00 0.00 0.00 0.00

3.46 0.00 7.32 0.00 0.00 0.00 0.00

0.00 7.32 0.00 5.73 0.00 7.56 0.00

0.00 0.00 5.73 0.00 0.00 0.00 2.13

0.00 0.00 0.00 0.00 0.00 5.25 0.00

0.00 0.00 7.56 0.00 5.25 0.00 3.96

0.00 0.00 0.00 2.13 0.00 3.96 0.00

The shortest distance from start to end is: 18.645
0 -> 1 -> 2 -> 3 -> 6
```

(program skompilowany z opcją -DDEBUG)

Wnioski i spostrzeżenia

Przechodzenie labiryntu pozwala na lepsze zrozumienie teorii grafów w informatyce oraz pozwala na zrozumienie działania algorytmów. Dużą zaletą tego zadania jest fakt, że do jego rozwiązania można użyć różnych algorytmów i nie trzeba się w nich ograniczać. Znaczącym minusem natomiast jest to, że w języku C jest to trudne zadanie i niestety zamiast skupiać się na praktycznym użyciu algorytmu, większym problemem było napisanie odpowiedniego kodu w tym języku.