



**Wydział
Elektryczny**

POLITECHNIKA WARSZAWSKA

JĘZYKI I METODY PROGRAMOWANIA 2

Dokumentacja projektowa: labfinder

Oliwia Pawelec, Jakub Żebrowski

prowadzący zajęcia:

Dr inż. Radosław Roszczyk

08.03.2024, modyfikacja: 22.04.2024

Cel projektu

Celem projektu jest opracowanie programu o nazwie **labfinder**, znajdującego najkrótszą drogę od wybranego punktu wejściowego do wybranego punktu wyjściowego przez labirynt.

Program działa w trybie nieinteraktywnym (wsadowym). Labirynt, przez który będzie opracowana ścieżka, będzie znajdować się w pliku wyznaczonym przez użytkownika w argumentach wywołania programu. Wynik programu będzie umieszczany na konsoli lub w pliku, który użytkownik wyznaczy w kolejnym argumencie wywołania.

Na projekt został nałożony limit pamięciowy 512 kB w czasie całego działania. Również projekt musi obsługiwać pliki binarne, których specyficzna struktura została opisana w założeniach projektowych napisanych przez prowadzących program zajęć. Jej opis jest dostępny w ostatniej sekcji tego dokumentu: *Dodatek: struktura specyficznego pliku binarnego*. Warto dodać, że zauważone zostały tam pewne błędy - ich opis oraz sposób na obejście ich znajduje się w sekcji *Założenia projektowe*.

Opis funkcjonalności

- **Wczytanie pliku:** program wczytuje plik, którego nazwę podaje użytkownik. Program sprawdza również czy plik istnieje - jeśli nie, daje komunikat o tym.
- **Odkodowanie wczytanego labiryntu:** jeśli użytkownik podał plik z kodowaniem binarnym, program wyluska z niego informacje na temat struktury labiryntu albo zwróci komunikat, że kodowanie jest nieznane/niepoprawne.
- **Walidacja struktury labiryntu:** program sprawdza, czy dany przez użytkownika labirynt jest poprawny: czy jest wejście/wyjście, równa liczba kolumn/wierszy etc.
- **Znalezienie ścieżki:** program znajduje najkrótszą ścieżkę przez labirynt od punktu startowego do punktu końcowego. Zapamiętuje przechodzoną ścieżkę.
- **Zapis ścieżki:** program zapisuje wybraną przez siebie ścieżkę (w formacie opisanym w rozdziale *Wykorzystywane pliki*) do wskazanego przez użytkownika pliku wyjściowego - jeśli plik o takiej nazwie nie istnieje to go tworzy - albo wypisuje wszystko na konsolę, jeśli użytkownik nie wybrał pliku docelowego.

Wykorzystywane pliki

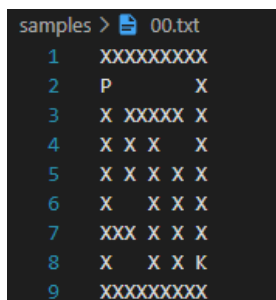
Plik wejściowy: obsługiwane są dwa formaty plików: tekstowy i binarny

1. W formacie **tekstowym**, labirynt powinien występować w formacie, który zawiera definicje punktów reprezentujących odpowiednio:

- P – punkt wejścia do labiryntu
- K - punkt wyjścia z labiryntu,
- X - ściana
- spacja - miejsce, po którym można się poruszać

Plik ten składać się powinien z pojedynczych znaków, których położenie x, y reprezentuje faktycznie położenie w labiryncie.

W pliku mogą pojawić się puste linie między wierszami/na końcu pliku. Będą one ignorowane ignorowane.



```
samples > 00.txt
1  XXXXXXXXX
2  P      X
3  X XXXXX X
4  X X X  X
5  X X X X X
6  X  X X X
7  XXX X X X
8  X  X X K
9  XXXXXXXXX
```

Figure 1: Przykładowy wygląd pliku wejściowego w formacie **tekstowym**

2. W formacie **binarnym**, labirynt powinien wpasowywać się strukturą w narzucony standard, opisany w ostatniej sekcji dokumentacji *Dodatek: struktura specyficznego pliku binarnego*

Plik wyjściowy: zawiera najkrótszą trasę przez wybrany labirynt, opisaną w formacie listy wykonanych kroków, na przykład:

```
START
FORWARD 1
TURNLEFT
FORWARD 4
TURNRIGHT
FORWARD 3
STOP
```

Tu również obsługiwane są oba (**tekstowe** i **binarne**) rodzaje plików. Kodowanie plików zachodzi w analogiczny sposób co do obsługi pliku wsadowego. Jeśli

użytkownik zechce rozwiązanie w formie **tekstowej**, do pliku wyjściowego wypisana zostanie lista kroków. Jeśli jednak w formie **binarnej**, do pliku wyjściowego zostanie zapisany zakodowany w formie binarnej labirynt, oraz pod nim zakodowaną (również binarne) listę kroków do przejścia najkrótszej trasy.

Założenia projektowe

Przyjęte zostały pewne założenia, które warto wymienić przed dalszym opisem wykorzystanych algorytmów i modułów.

1. labirynt:

- wymiary labiryntu (licząc same komórki, bez doliczania ścianek między nimi) spełniają nierówność: $1 \leq x, y \leq 1024$
- między dowolną parą stojących obok siebie rzędów/kolumn musi wystąpić chociaż 1 ścianka
- grubość każdej ścianki wynosi 1
- wejście i wyjście do labiryntu znajdują się na zewnętrznym obramowaniu labiryntu

2. plik z rozwiązaniem:

Zwiększony został rozmiar `steps` w Sekcja nagłówkowa `rozwiązania` z 8 bitów na 32. Tak samo rozmiar `Counter`. Taka zmiana musiała zajść ze względu na to, że liczba kroków może przekroczyć 255, tak samo jak i liczba kroków w jednym kierunku - może dojść do 2048.

Wykorzystane algorytmy

Przechowywanie labiryntu

Labirynt jest przechowywany w dwóch macierzach bitowych. Są to dwie dwuwymiarowe tablice o rozmiarach 1028x256 i 1028x128 bajtów. Na potrzeby implementacji algorytmu wyszukiwania ścieżki, zwiększony został rozmiar macierzy przechowującej strukturę do 1152x256 bajtów.

1 bajt składa się z 8 bitów. Labirynt można skompresować poprzez przechowywanie informacji o jednej komórce na jednym bicie. Wówczas wymagana pamięć na przechowywanie zmniejszy się ośmiokrotnie.

Pierwsza macierz przechowuje informacje na temat struktury labiryntu. Każda komórka zawiera informację o ściankach między tą komórką, a komórką: pod i obok z prawej strony. W ten sposób zachowana zostanie cała struktura labiryntu z pliku, bez niepotrzebnego, podwójnego zapisywania informacji o ściankach.

Druga macierz zawiera binarne stany komórek: czy została odwiedzona (1) czy jeszcze nie (0). Komórek maksymalnie może być 1024x1024, czyli tym sposobem można skompresować labirynt do rozmiarów 1024x128 bajtów (ok. 131 KB).

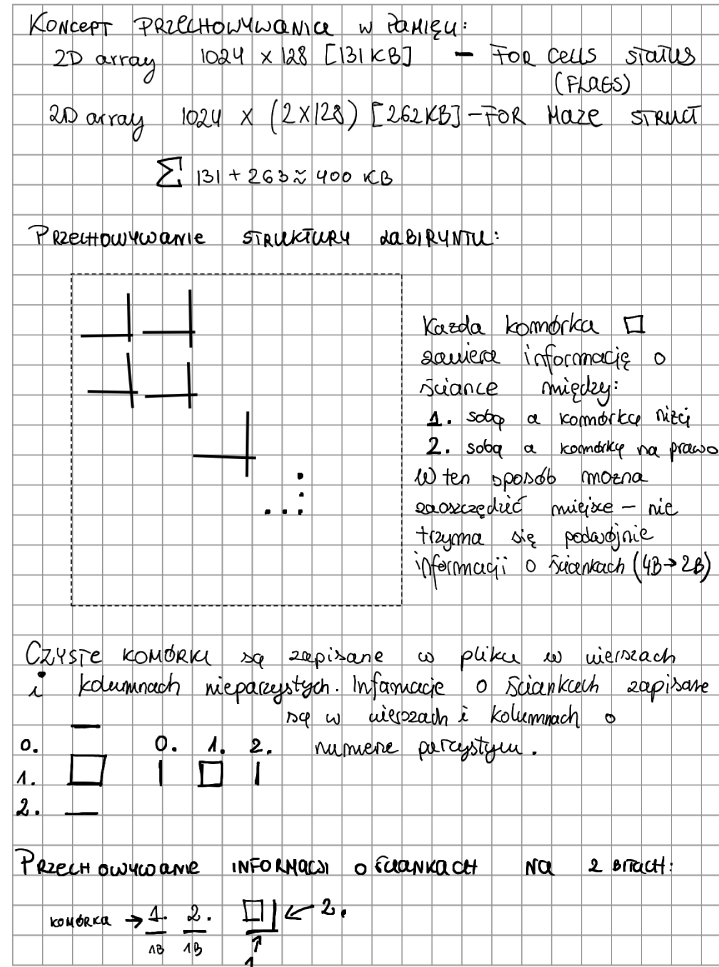


Figure 2: Zwizualizowany i opisany koncept przechowywania labiryntu

Algorytm przechodzenia

Do znalezienia najkrótszej ścieżki został użyty algorytm BFS (ang. *Breadth First Search*). Operuje on na drugiej macierzy 1024x128 i kolejnie obok, do której kolejno są dodawani sąsiedzi. Zmieniane są kolejno flagi komórek

odwiedzonych na '1'. Po znalezieniu punktu wyjściowego, ustawia resztę flag (nie uwzględnionych w trasie) na '0'. Na podstawie tego ustalana jest ścieżka - zapisywana do wskazanego pliku wyjściowego.

Wykorzystywana kolejka znajduje się w pliku tymczasowym - w najgorszym przypadku, znaleziona ścieżka może być krzywą Hilberta, co znacznie przekroczy założony limit pamięciowy.

Podział na moduły

Program został podzielony na moduły:

- obsługa wejścia/wyjścia: obsługa plików wejściowych/wyjściowych, sprawdzanie błędów
- operacje na bitach: ułatwienie operacji na bitach (dostanie się do konkretnego bita, ustawienie, sprawdzenie jego wartości)
- odkodowywanie plików: odczyt struktury labiryntu z pliku, zapis tej struktury w programie
- najkrótsza ścieżka: znajdowanie najkrótszej ścieżki przez podany labirynt
- zakodowywanie plików: zapis struktury labiryntu do pliku wraz z wybraną trasą przez ten labirynt

Obsługa wejścia/wyjścia

W tym module znajduje się funkcja sprawdzająca dodatkowe argumenty - wyrzuca komunikat o błędzie kiedy np. jakichś argumentów będzie brakować.

Program wstępnie zaakceptuje wejście, w którym użytkownik zapewni ścieżkę do pliku wejściowego oraz jego kodowanie. Informacje o pliku wyjściowym i jego kodowaniu są opcjonalne. Jeśli użytkownik nie poda ścieżki do tego pliku, wszystko będzie wypisane na `stdout` w kodowaniu tekstowym, niezależnie od wybranego przez użytkownika kodowania wyjściowego. Z kolei gdy zapewni ścieżkę do pliku bez informacji o zakodowaniu, do wybranego pliku zostanie zapisane rozwiązanie w kodowaniu tekstowym.

```
while ((opt = getopt(argc, argv, ":i:o:c:d:")) != -1) {
    switch (opt) {
        case 'i':
            in_file = optarg;
            break;
        case 'o':
            out_file = optarg;
            break;
```

```

        case 'c':
            in_code = optarg;
            break;
        case 'd':
            out_code = optarg;
            break;
        default:
            return help(argv[0]), EXIT_FAILURE;
    }
}

```

Na `stdout` powinien wyświetlić się komunikat pomocy, kiedy podane argumenty będą niepoprawne.

```

void help(char* file) {
    printf("UZYSKIE PROGRAMU:\n\t%s -i <plik_wejscowy> -o  

        <plik_wyjsciowy> -c <kod_wejscowy> -d <kod_wyjsciowy>\n",
        file);
}

```

Operacje na bitach

Moduł będzie składał się z makr zajmujących się operacjami na bitach: sprawdzeniem wartości danego bita na `a`, `b` pozycji w macierzy, ustawieniem jego wartości na konkretną `c`, zmienieniem jej na przeciwną, sprawdzeniem ile bitów mieści się w podanej zmiennej `a`.

```

#define GETBIT(a, b)
#define SETBIT(a, b, c)
#define FLIPBIT(a, b)
#define BITSIZEOF(a)

```

Odkodowywanie plików

Moduł najpierw zajmuje się sprawdzaniem czy kodowanie jest poprawne - poniżej przykładowy fragment kodu, który sprawdza, czy liczba wierszy jest wszędzie taka sama.

```

char is_file_coding_valid(FILE* input_file, char coding_type) {
    ...
    if (rows_count != rows[1])
        return 1; // invalid
    ...
    return 0; // valid
}

```

Moduł musi zawierać funkcję transformującą zakodowany labirynt za labirynt w formie macierzy bitowej `maze[256][256]` i `maze[128][128]`.

```
void decode_file(FILE* input_file, char coding_type) {
    ...
}
```

Najkrótsza ścieżka

W module pojawia się funkcja z implementacją BFS.

```
void bfs(char maze_struct[256][256], char maze_flags[128][128]) {
    // forma kolejki, do której będzie się dodawać kolejne sąsiednie komórki
    while (queue.size != 0) {
        ...
    }
    ...
}
```

Na podstawie labiryntu z flagami, zostanie opracowana ścieżka:

```
void create_path(path* some_path, char maze_2[128][128]) {
    ...
}
```

Zakodowywanie plików

W tym module zapisywana jest trasa do plików docelowych. Jeśli kodowanie pliku docelowego to tekstowe, plik z krokami jest bezpośrednio przepisywany do wyjścia.

```
byte encode_txt( FILE* output_file, FILE* steps_file ) {
    fprintf(output_file, "START\n");
    char c;
    while ( (c = fgetc(steps_file)) != EOF )
        fprintf(output_file, "%c", c);
    fprintf(output_file, "STOP\n");
    return 0;
}
```

Z kolei jeśli format docelowy jest binarny, labirynt jest zakodowywany według podanego standardu pliku binarnego. Następnie dopisywany jest nagłówek z drogą przez labirynt.

```
byte encode_binary(FILE* output_file, FILE* steps_file,
    byte maze_struct[][256], bit_pair* maze_size,
    maze_cord* in_cord, maze_cord* out_cord ) {...}
```

Argumenty wywołania programu

Program może być wywoływany w trzy sposoby:

labfinder przedstawia informacje o poprawnym wywołaniu programu oraz o jego działaniu.

labfinder -i <plik wejściowy> -o <plik wyjściowy> -c <kodowanie wejścia> -d <kodowanie wyjścia>

wczytuje podany plik wejściowy w kodowaniu określonym jako t (**tekstowe**) albo b (**binarne**), a następnie wypisuje listę kroków do wskazanego pliku wyjściowego w podanym kodowaniu

labfinder -i <plik wejściowy> -c <kodowanie wejścia>

wczytuje podany plik wejściowy w kodowaniu określonym jako t (**tekstowe**) albo b (**binarne**), a następnie wypisuje listę kroków na **stdout** - podane tu kodowanie pliku wyjściowego będzie ignorowane - domyślnie na **stdout** wypisywane są kroki w kodowaniu tekstowym

Program nie wymaga dodania rozszerzenia do nazw plików wejściowych i wyjściowych, wymaga jedynie podania rodzaju kodowania pliku wsadowego.

Obsługiwane są jedynie dwa rodzaje kodowania:

- tekstowe (znak t)
- binarne (znak b)

Inne symbole będą odrzucane przez program.

Komunikaty błędów

Program będzie komunikował błędy, gdy:

- będzie brakowało argumentów: np. z nazwą pliku wsadowego, z informacją o kodowaniu pliku wsadowego - informacja o pliku wyjściowym i o jego kodowaniu może zostać pominięta - wówczas na konsolę zostanie wypisana lista kroków przejścia w formacie tekstowym
- argumenty wywołania nie będą poprawne (np. wskazane przez użytkownika pliki nie będą istniały)
- kodowanie pliku z labiryntem nie będzie w poprawnym formacie
- labirynt będzie uszkodzony: będzie brakowało punktu start/koniec lub będzie ich więcej niż po jednym, nie będzie istniała ścieżka prowadząca z punktu początkowego do końcowego, liczba wierszy/kolumn będzie zmienna

Testowanie

Program powinien być w stanie przejść testy wykonane przy użyciu programu *valgrind*, również przy użyciu komendy

```
/usr/bin/time -v
```

Dodatek: struktura specyficznego pliku binarnego

Dane wejściowe w formacie binarnym podzielone są na 4 główne sekcje:

1. Nagłówek pliku
2. Sekcja kodująca zawierająca powtarzające się słowa kodowe
3. Nagłówek sekcji rozwiązania
4. Sekcja rozwiązania zawierające powtarzające się kroki które należy wykonać aby wyjść z labiryntu

Sekcja 1 i 2 są obowiązkowe i zawsze występują, sekcja 3 oraz 4 są opcjonalne. Występują jeśli wartość pola *Solution Offset* z nagłówka pliku jest różna od 0.

Nagłówek pliku:

Nazwa pola	Wielość w bitach	Opis
File Id	32	Identyfikator pliku: 0x52524243
Escape	8	Znak ESC: 0x1B
Columns	16	Liczba kolumn labiryntu (numerowane od 1)
Lines	16	Liczba wierszy labiryntu (numerowane od 1)
Entry X	16	Współrzędne X wejścia do labiryntu (numerowane od 1)
Entry Y	16	Współrzędne Y wejścia do labiryntu (numerowane od 1)
Exit X	16	Współrzędne X wyjścia z labiryntu (numerowane od 1)
Exit Y	16	Współrzędne Y wyjścia z labiryntu (numerowane od 1)
Reserved	96	Zarezerwowane do przyszłego wykorzystania
Counter	32	Liczba słów kodowych
Solution Offset	32	Offset w pliku do sekcji (3) zawierającej rozwiązanie
Separator	8	słowo definiujące początek słowa kodowego – mniejsze od 0xF0
Wall	8	słowo definiujące ścianę labiryntu
Path	8	słowo definiujące pole po którym można się poruszać
Podsumowanie	420	Sumarycznie nagłówek ma rozmiar 40 bajtów

Słowa kodowe:

Nazwa pola	Wielość w bitach	Opis
Separator	8	Znacznik początku słowa kodowego
Value	8	Wartość słowa kodowego (Wall / Path)
Count	8	Liczba wystąpień (0 – oznacza jedno wystąpienie)

Sekcja nagłówkowa rozwiązania

Nazwa pola	Wielość w bitach	Opis
Direction	32	Identyfikator sekcji rozwiązania: 0x52524243
Steps	8	Liczba kroków do przejścia (0 – oznacza jeden krok)

Krok rozwiązania:

Nazwa pola	Wielość w bitach	Opis
Direction	8	Kierunek w którym należy się poruszać (N, E, S, W)
Counter	8	Liczba pól do przejścia (0 – oznacza jedno pole)

Pola liczone są bez uwzględnienia pola startowego.