

# JiMP 2 sprawozdanie z projektu w C

Paweł Skierkowski, Łukasz Jarzecki

April 2024

# Contents

<b>1</b>	<b>Wstęp</b>	<b>3</b>
1.1	Problem zadania . . . . .	3
1.2	Sposób rozwiązania . . . . .	3
<b>2</b>	<b>Implementacja</b>	<b>6</b>
2.1	Modularność . . . . .	6
2.2	Działanie programu . . . . .	8
2.3	Funkcjonalność . . . . .	9
2.4	Uruchamianie programu . . . . .	10
<b>3</b>	<b>Testy</b>	<b>10</b>
3.1	Labirynt 100x100 . . . . .	10
3.2	Labirynt 1024x1024 . . . . .	10
3.3	Labirynt binarny (duży labirynt binarny z ISOD) . . . . .	11
3.4	Labirynt $maze_{turns_1}1024_1023$ . . . . .	11
3.5	Labirynt $maze_{forks_1}1023_1023$ . . . . .	11
3.6	Labirynt $maze_{forks_5}12_511$ . . . . .	11
<b>4</b>	<b>Wnioski</b>	<b>11</b>

# 1 Wstep

Celem projektu było stworzenie programu, który umożliwi znaleźć wyjścia z labiryntu wczytanego przez użytkownika. Labirynt wprowadzany jest z pliku tekstowego lub binarnego.

## 1.1 Problem zadania

Głównym problemem jest implementacja algorytmu, który umożliwi znalezienie ścieżki, która przemierzając się po labiryncie znajduje drogę między punktem P (początkiem) a K (końcem). Plik tekstowy składa się ze znaków "X" oznaczających ściany oraz " " oznaczających przejście, jak i opisanych wcześniej 'P' i 'K'. Natomiast plik binarny składa się z poleceń opisujących budowę labiryntu.

Wymaganiem, które stworzyło dodatkową trudność dla projektu było wymagane ograniczenie zużycia przez program pamięci, ustalonym na 512kB w trakcie czasu działania. Wyzwanie to wymaga głębokiego zrozumienia zarówno algorytmów, jak i niskopoziomowych aspektów zarządzania zasobami w programowaniu. Wynikiem działania programu, ma być lista kroków potrzebnych do przejścia przez labirynt w formie :

START

FORWARD 8

TURN LEFT

STOP

## 1.2 Sposób rozwiązania

Do rozwiązania problemu zastosowaliśmy algorytm Trémaux.

Algorytm ten działa, poprzez stosowanie się do poniższych zasad:

- Gdy znajdujesz się na początku ścieżki, nie ważne czy jest to wejście na skrzyżowanie, czy też zejście z niego zostaw znacznik.(patrz rys. 1.)
- Jeżeli ścieżka jest oznaczona 2 razy, to droga ta zostaje odcięta (na rysunku oznaczenie pojedyncze to '.', a podwójne to 'X').
- Kiedy jesteś na skrzyżowaniu wybierz jedno z dostępnych wyjść.

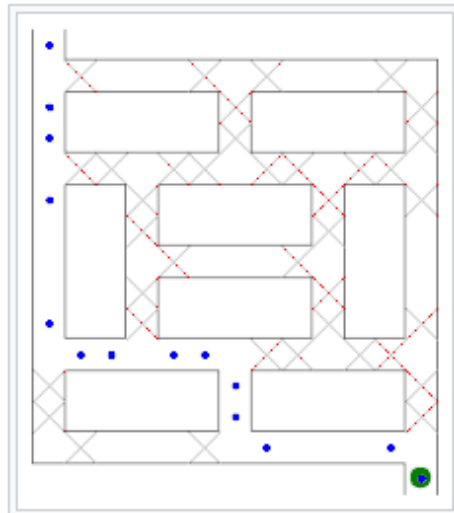


Figure 1: Algorytm Trémaux

- Jeżeli ze skrzyżowania wychodzi ścieżka jeszcze nie oznaczona wybierz tą ścieżkę (niema znaczenia która)
- Jeżeli przejście, których przyszedłeś nie jest oznaczone 2 razy to wróć tą samą drogą (pomaga to z wykrywaniem ślepych zaułków i petli)
- Wybierz drogę z najmniejszym możliwym oznaczeniem

Po zakończeniu oznaczania ścieżek i znalezieniu wyjścia, należy połączyć wszystkie przejścia oznaczone raz. Prowadzi to do znalezienia właściwej drogi.

W naszym rozwiązaniu przyjęliśmy strategię oznaczania ścieżek, która zakłada że pojedyncze oznaczenie to '1', natomiast podwójne to '2'. Przykład rozwiązania prostego labiryntu poniżej.

```

XXXXXXXXXXXXXXXXXXXXX
P 2 X X X X X
X1X2X X X XXX XXX X X
X X X X X X X
X1XXXXXXX X XXXXXX X
X 1 1 X X X X X
XXXXXXXX1X XXX X X XXX
X 1 1 X X X X X
X1XXX1X1XXX XXXXXX X
X X X X X X
X X2X1X1X XXXXXXXX X
X X 2 1 X X X
X X2XXXXXXX XXXXXX
X X 2 X 1 1 X X X
X XXX2X1XXX1X XXX X X
X X 2 X X X X X
X1XXXX1X2X1XXXXXXX X
X 1 1 X X 1 X X
X2XXXXXX2XXX1X XXXXX
X 2 2 2 X 1 1 K
XXXXXXXXXXXXXXXXXXXXX

```

Figure 2: Oznaczenie ścieżki

Jest to pierwszy krok do rozwiązania, następnym krokiem jest połączenie je-  
dynek, oraz wypisanie instrukcji.

```

XXXXXXXXXXXXXXXXXXXXX
..2 X X X X X
X.X2X X X XXX XXX X X
X.X X X X X X
X.XXXXXX X XXXXXX X
X.....X X X X X
XXXXXXXX.X XXX X X XXX
X....X.X X X X
X.XXX.X.XXX XXXXXX X
X.X X.X. X X
X.X2X.X.X XXXXXXXX X
X.X 2...X X X
X.X2XXXXXXX XXXXXX
X.X 2 X.....X X X
X.XXX2X.XXX.X XXX X X
X.X 2 X.X.X X X
X.XXXXX.X2X.XXXXXX X
X.....X X...X X
X2XXXXXX2XXX.X XXXXX
X 2 2 2 X.....K
XXXXXXXXXXXXXXXXXXXXX

```

Figure 3: Połączenie punktów

Poprawna ścieżka jest w tym momencie oznaczona kropkami. Lista kroków  
wypisana do pliku kroki.txt:

```

START
FORWARD 1
TURN RIGHT
FORWARD 4
TURN RIGHT
FORWARD 6
TURN RIGHT
FORWARD 6
TURN LEFT
FORWARD 2
TURN RIGHT
FORWARD 4
TURN RIGHT
FORWARD 4
TURN LEFT
FORWARD 10
TURN RIGHT
FORWARD 6
TURN LEFT
FORWARD 4
TURN LEFT
FORWARD 4
TURN RIGHT

```

Figure 4: Lista kroków

## 2 Implementacja

### 2.1 Modularność

Kluczowym aspektem realizacji projektu jest rozdzielenie go na moduły, co umożliwiło efektywną pracę i ułatwiło zarządzanie kodem. Dokonałiśmy podziału na następujące moduły:

- **Algorytm** (algorithm.c, algorithm.h): w module tym zaimplementowany jest algorytm, który odpowiada za główne działanie programu. Poza zaznaczaniem drogi do wyjścia zawiera funkcje wykorzystywane w innych modułach np. podstawowy kierunek przemieszczania się, zatrzymanie działania programu.
- **Odczytywanie ścieżki** (read\_path.c, read\_path.h): moduł ten odpowiada za odnalezienie oznaczonej przez główny algorytm ścieżki, oraz wypisanie kroków potrzebnych do przejścia.
- **Operacje na plikach** (fhandling.c, fhandling.h): w tym module odbywa się wczytywanie pliku, kopiowanie pliku rozwiązania, ustalanie pozycji, zresetowanie obecnej pozycji w pliku, znalezienie koordynatów wejścia i wyjścia z labiryntu. Stworzone są w niej również funkcje 'replaceChr' czyli zamiana znaku w pliku znajdującego się na ustalonej pozycji, na znak podany w wywołaniu funkcji, oraz 'getChr' która zwraca znak z pliku na żądanej pozycji.
- **MAIN** (main.c, macro.h): main.c jest swego rodzaju spiciem wszystkich funkcjonalności programu w jedno, natomiast macro.h odpowiada za do-

anie bibliotek oraz powołanie struktur zawierających dane o labiryncie oraz kierunkach ruchu.

- **Plik binarny**(binary.c, binary.h): ten moduł programu odpowiada za roszyfrowywanie pliku binarnego, a następnie kodowanie rozwiązania w postaci komend, jakie znajdujemy w pliku binarnym.

## 2.2 Działanie programu

Program rozpoczyna swoje działanie, od stworzenia kopii pliku (copy\_file) z labiryntem, na którym później sam będzie pracował. Dzięki temu plik bazowy z labiryntem nie zostaje zniszczony i może być wykorzystywany wiele razy, bez konieczności generowania nowego labiryntu. Kolejnym etapem jest określenie punktów bazowych labiryntu i zapisanie ich do struktury, a następnie wypisanie ich na ekran konsoli w celu podania podstawowych informacji o labiryncie.

```
resetPointer(f);
struct maze s = mazeData(f);
printf("x - %d, y - %d\nstart - %d/%d\nkoniec - %d/%d\n", s.size_x, s.size_y, s.start_x, s.start_y, s.end_x, s.end_y);
```

Figure 5: Podstawowe informacje

Następnie wywoływana jest funkcja 'alg' znajdująca się w module algorithm.c, która za pomocą petli while przemieszcza się po pliku, sprawdzając czy nie jest to koniec oraz oznacza przejścia z wykorzystaniem wcześniej opisanych funkcji wyszukiwania oraz zamieniania znaków.

```
printf("x - %d, y - %d\nstart - %d/%d\nkoniec - %d/%d\n", s.size_x, s.size_y, s.start_x, s.start_y, s.end_x, s.end_y);

alg(f, s);

resetPointer(f);
```

Figure 6: Wywołanie algorytmu

```
int tsj = 0; // time since juncture

while (!isFinished(in, maz, mov)) {

    //sleep(2000);

    // jest prosta droga
    if (!passageType(in, maz, mov)) {
        mov.x = mov.y;
```



Funkcja po oznaczeniu ścieżki zostaje przekazana do modułu `read_path.c`, w którym wykonywana jest funkcja `read_path()`.

```
resetPointer(f);
struct maze m = mazeData(f);
read_path(f, m);
```

Figure 7: Wywołanie odczytywania ścieżki

Zostaje tutaj zastosowane wyszukiwanie kolejnych jedynek w zależności od odpowiedniego kierunku.

```
while (!last_step(in,x,y,m))
{
    switch (kierunek) {
    case 'r':
        count_steps = 0;
        while (getChr(in, y, x, m) != 'X' && getChr(in, y, x, m) != '2' && getChr(in, y, x, m) != 'K') {
            replaceChr(in, y, x, '.', m);
```

Figure 8: Petla przechodząca po pliku i odczytująca ścieżkę

W tej funkcji odbywa się równolegle wypisywanie kolejnych kroków, które są ostatecznym wynikiem działania programu. Wszystkie wypisane funkcje bazują na tych, znajdujących się w module `fhandling.c`.

## 2.3 Funkcjonalność

- Efektywne znajdowanie ścieżki: Użycie algorytmu Trémaux aby znaleźć drogę przez labirynt.
- Dynamiczne wczytywanie labiryntów: Możliwość wczytania dowolnego labiryntu z pliku, z automatycznym rozpoznaniem rozmiaru oraz lokalizacji startu i końca.
- Eksport wyników: Możliwość zapisania wykonanych kroków do pliku (`kroki.txt`), co ułatwia analizę i prezentację rozwiązania.

## 2.4 Uruchamianie programu

Uruchamianie programu rozpoczynamy od dostarczenia pliku z labiryntem do katalogu, w którym znajduje się plik wykonywalny. Następnie, aby uruchomić program z zadany plikiem należy:

1. Otwórz terminal w systemie Linux.
2. Przejdź do katalogu zawierającego pliki z rozwiązaniem.
3. Skompiluj program używając polecenia `'make'`
4. Wywołaj program stosując komendę `./program -n nazwa.pliku -t typ.pliku` (parametru mogą być podane w dowolnej kolejności, ponieważ zastosowaliśmy funkcję `getopt`) dozwolonymi parametrami są pliki rozszerzeniach `.bin` oraz `.txt` wpisane po `'-n'` oraz rozszerzenia tych plików w formacie `'bin'` lub `'txt'` podane po `'-t'`. Jeżeli nie podamy rozszerzenia (np `-t txt`) program założy, że podaliśmy plik o rozszerzeniu `.txt`, maksymalna długość nazwy pliku to 50 znaków.

## 3 Testy

Testując oprogramowanie badaliśmy zarówno czas wykonania, jak i zużycie pamięci dla labiryntów o różnej złożoności. Różniły się one typem pliku (tekstowy lub binarny) oraz wielkością (od 100x100 do 1024x1024). Do testów używaliśmy komend `'memusage'` i `'time'`.

### 3.1 Labirynt 100x100

Czas wykonania: 0.026s

Zużycie pamięci: 9,156B

### 3.2 Labirynt 1024x1024

Czas wykonania: 3.14s

Zużycie pamięci: 9,156B

### 3.3 Labirynt binarny (duży labirynt binarny z ISOD)

Czas wykonania: 0.4s

Zużycie pamięci: 13,754B

### 3.4 Labirynt $\text{maze}_{turns_1}024_1023$

Czas wykonania: 1m 6.29s

Zużycie pamięci: 9,186B

### 3.5 Labirynt $\text{maze}_{forks_1}023_1023$

Czas wykonania: 54.7s

Zużycie pamięci: 9,186B

### 3.6 Labirynt $\text{maze}_{forks_5}12_511$

Czas wykonania: 13.5s

Zużycie pamięci: 9,186B

## 4 Wnioski

Realizacja tego projektu umożliwiła nam poszerzenie naszej znajomości środowiska C oraz nasze umiejętności współpracowania nad wspólnym projektem. Największym problemem, z którym musieliśmy się zmierzyć było oczywiście ograniczenie zużycia pamięci. Okazało się, że najważniejszym elementem było dobranie odpowiedniego sposobu, czy algorytmu.

Wybraliśmy algorytm Trémaux ponieważ wydawał się on nam nietypowy oraz ciekawy. Praca nad rozwiązaniem była interesująca również z powodu graficznej elegancji, jaką prowadzi za sobą ten algorytm.

Wspólnie prace rozpoczęliśmy na systemie Windows, a następnie przenieśliśmy ją na system Linux poznaliśmy dzięki temu różnice systemowe, które jeszcze bardziej pogłębiły naszą wiedzę.

Podczas pracy nauczyliśmy się efektywnej współpracy i komunikacji. Projekt ukazał nam realia złożoności i konsekwencji naszych wyborów. Musieliśmy odpowiednio wykorzystać język C aby stworzyć swój mały 'system' funkcji globalnych , który pomógł nam rozwiązać program w lepszy sposób.