JiMP 2 sprawozdanie z projektu w C

Paweł Skierkowski, Łukasz Jarzecki

April 2024

Contents

1 Wstep

Celem projektu było stworzenie programu, który umożliwia 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 przemieszczajac sie po labiryncie znajduje droge miedzy punktem P (poczatkiem) a K (końcem). Plik tekstowy składa sie ze znaków "X" oznaczajacych ściane oraz " " oznaczajacych przejście, jak i opisanych wcześniej 'P' i 'K'. Natomiast plik binarny składa sie z poleceń opisujacych budowe labiryntu.

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

START

FORWARD 8

TURN LEFT

STOP

1.2 Sposób rozwiazania

Do rozwiazania problemu zastosowaliśmy algorytm Trémaux.

Algorytm ten działa, poprzez stosowanie sie do poniższych zasad:

- Gdy znajdujesz sie na poczatku 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 odcieta (na rysunku oznaczenie pojedyncze to '.', a podwójne to 'X').
- Kiedy jesteś na skrzyżowaniu wybierz jedno z dostepnych wyjść.

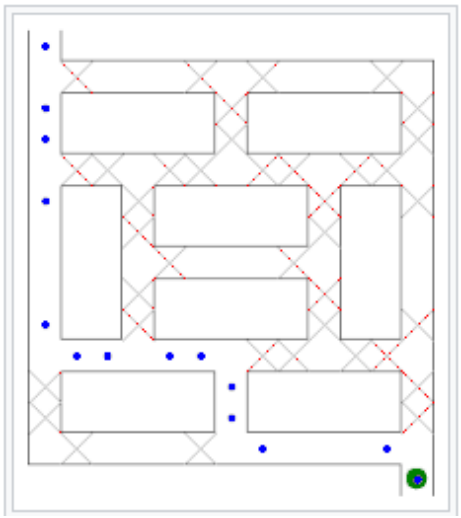

Figure 1: Algorytm Trémaux

- Jeżeli ze skrzyżowania wychodzi ścieżka jeszcze nie oznaczona wybierz ta ścieżke (niema znaczenia która)
- Jeżeli przejście, których przyszedłeś nie jest oznaczone 2 razy to wróć ta sama droga (pomaga to z wykrywaniem ślepych zaułków i petli)
- Wybierz droge z najmniejszym możliwym oznaczeniem

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

W naszym rozwiazaniu przyjeliśmy strategie oznaczania ścieżek, która zakłada że pojedyncze oznaczenie to '1', natomiast podwójne to '2'. Przykład rozwiazania prostego labiryntu poniżej.

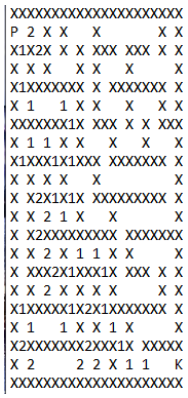

Figure 2: Oznaczenie ścieżki

Jest to pierwszy krok do rozwiazania, nastepnym krokiem jest połaczenie jedynek, oraz wypisanie instrukcji.

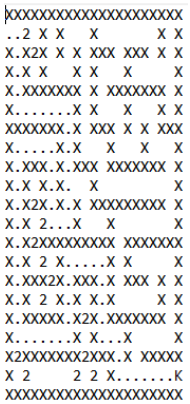

Figure 3: Połaczenie punktów

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

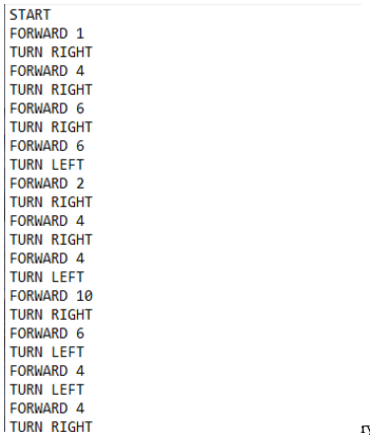

Figure 4: Lista kroków

2 Implementacja

2.1 Modularność

Kluczowym aspektem realizacji projektu jest rozdzielenie go na moduły, co umożliwiło efektywna prace i ułatwiła zarzadzanie kodem. Dokonaliśmy podziału na nastepujace 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 sie, 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 sie wczytywanie pliku, kopiowanie pliku rozwiazania, ustalanie pozycji, zresetowanie obecnej pozycji w pliku, znalezienie koordynatów wejścia i wyjścia z labiryntu. Stworzone sa w niej również funkcje 'replaceChr' czyli zamiana znaku w pliku znajdujacego sie na ustalonej pozycji, na znak podany w wywołaniu funkcji, oraz 'getChr' która zwraca znak z pliku na żadanej pozycji.
- **MAIN** (main.c, macro.h): main.c jest swego rodzaju spieciem wszystkich funkcjonalności programu w jedno, natomiast macro.h odpowiada za do-

danie bibliotek oraz powołanie struktur zawierajacych dane o labiryncie oraz kierunkach ruchu.

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

2.2 Działanie programu

Program rozpoczyna swoje działanie, od stworzenia kopi pliku (copy_file) z labiryntem, na którym później sam bedzie pracował. Dzieki 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 nastepnie 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

Nastepnie wywoływana jest funkcja 'alg' znajdujaca sie w module algorith.c, która za pomoca petli while przemieszcza sie po pliku, sprawdzajac czy nie jest to koniec oraz oznacza przejścia z wykorzystaniem wcześniej opisanych funkcji wyszukiwania oraz zamieniania znaków.

```
alg(f, s);
```

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)) {
```

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 przechodzaca po pliku i odczytujaca ścieżke

W tej funkcji odbywa sie równolegle wypisywanie kolejnych kroków, które sa ostatecznym wynikiem działania programu. Wszystkie wypisane funkcje bazuja na tych, znajdujacych sie w module fhandling.c.

2.3 Funkcjonalność

- Efektywne znajdowanie ścieżki: Użycie algorytmu Trémaux aby znaleźć droge 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 analize i prezentacje rozwiazania.

2.4 Uruchamianie programu

Uruchamianie programu rozpoczynamy od dostarczenia pliku z labiryntem do katalogu, w którym znajduje sie plik wykonywalny. Nastepnie, aby uruchomić program z zadanym plikiem należy:

1. Otwórz terminal w systemie Linux.
2. Przejdź do katalogu zwierajacego pliki z rozwiazaniem.
3. Skompiluj program używajac polecenia 'make'
4. Wywołaj program stosujac komende ./program -n nazwa.pliku -t typ.pliku (parametru moga być podane w dowolnej kolejności, ponieważ zastosowaliśmy funckje getopt) dozwolonymi parametrami sa 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

Testujac oprogramowanie badaliśmy zarówno czas wykonania, jak i zużycie pamieci dla labiryntów o różnej złożoność. Różniły sie one typem pliku (tekstowy lub binarny) oraz wielkościa (od 100x100 do 1024x1024). Do testów urzywaliśmy komend 'memusage' i 'time'.

3.1 Labirynt 100x100

Czas wykonania: 0.026s

Zużycie pamieci: 9,156B

3.2 Labirynt 1024x1024

Czas wykonania: 3.14s

Zużycie pamieci: 9,156B

3.3 Labirynt binarny (duży labirynt binarny z ISOD)

Czas wykonania: 0.34s

Zużycie pamieci: 13,724B

4 Wnioski

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

Wybraliśmy algorytm Trémaux ponieważ wydawał sie on nam nietypowy oraz ciekawy. Praca nad rozwiazaniem była interesujaca również z powodu graficznej elegancji, jaka prowadzi za soba ten algorytm.

Wspólnie prace rozpoczeliśmy na systemie Windows, a nastepnie przenieśliśmy ja na system Linux poznaliśmy dzieki temu różnice systemowe, które jeszcze bardziej pogłebiły nasza wiedze.

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