JIMP 2 Sprawozdanie z części projektu wykonanej przy pomocy języka C

Zespół 7

Spis treści

Wstęp

1. Opis programu.....3

2. Zasadnicza Trudność Zadania.....3

3. Obsługa programu przez użytkownika.....3

4. Wyjście programu.....4

Sposób Rozwiązania.....5

1. Wybrany do rozwiązania algorytm oraz opis jego działania..... 5

Podział Kodu.....6

Testy.....6

Wnioski.....6

Wstęp

1.Opis programu

Celem programu wytworzonego w ramach projektu oraz napisanego w języku C, było znalezienie drogi przez labirynt, maksymalnie o rozmiarach 1024x1024 liczonych po ścieżkach którymi możemy się poruszać. Trasa zaczyna się w punkcie oznaczonym literą P, kończy natomiast w punkcie oznaczonym literą K. Wynikiem działania programu powinna być lista kroków, które należy wykonać, aby wydostać się z labiryntu.

2.Zasadnicza Trudność Zadania

Głównym problemem podczas rozwiązywania zadania było sprawienie że program działać będzie przy limicie 512 KB pamięci użytej w jednym momencie. Wymagało to implementacji rozwiązań zmniejszających ich użycie, w naszym wypadku zdecydowaliśmy się na wykorzystanie przy większości operacji plików tymczasowych, w których można przechowywać dane, które wczytywane będą do zmiennych, co pozwala na operowanie jedynie niewielką ilością pamięci w jednym momencie, niestety przy wydłużeniu czasu działania programu.

3.Obsługa programu przez użytkownika

Program skompilować można przy pomocy skryptu Makefile, użytkownik musi jedynie po otworzeniu terminalu w folderze zawierającym projekt wpisać komendę make. Uruchomienie programu wymaga podania jako argumentu wywołania ścieżki do pliku z labiryntem. Przykład uruchomienia programu:

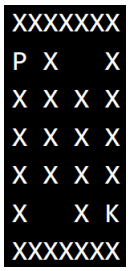
```
./maze-solver maze.bin
```

Plik taki można wygenerować za pomocą strony internetowej
będącej narzędziem dostarczonym na potrzeby projektu.

Plik ten może być formatu .txt. W takim pliku przyjęte zostały następujące oznaczenia:

- P – początek labiryntu (wejście)
- K – koniec labiryntu (wyjście)
- X – ściana
- Spacja – miejsce, po którym jest możliwość poruszania się.

Przykładowy wygląd pliku .txt prezentuje się w następujący sposób:



Plik może być również formatu .bin. Taki plik opisany jest w następujący sposób:

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.

4.Wyjście programu

Wynikiem działania programu jest stworzenie instrukcji kroków, które należy wykonać w celu osiągnięcia wyjścia z labiryntu. Instrukcje te zapisywane są do pliku tekstowego, zawsze nazywanego instructions.txt. Stosując opis wykorzystanego wcześniej przykładu pliku tekstowego, instrukcje wyglądają jak:

START => FORWARD 1 => TURN RIGHT => FORWARD 5 => TURN LEFT => FORWARD 2 => TURN LEFT => FORWARD 4 => TURN RIGHT => FORWARD 2 => TURN RIGHT => FORWARD 4 => STOP

Sposób Rozwiązania

1.Wybrany do rozwiązania algorytm oraz opis jego działania

Algorytmem który zdecydowaliśmy się wykorzystać do znalezienia drogi przez labirynt jest algorytm BFS (breadth-first search), który w naszej implementacji dokonał zasiedlenia pól labiryntu wartościami małych liter alfabetu angielskiego, od a do z, następnie po osiągnięciu z, następne pole zapisywane było ponownie jako a. Ponownie korzystając z podanego wcześniej jako przykład labiryntu, po zasiedleniu wyglądałby on jak:

```
XXXXXXX
PbXlmnX
XcXkXoX
XdXjXpX
XeXiXqX
XfghXrK
XXXXXXX
```

Aby zmniejszyć zużycie pamięci przez program kolejka przetwarzanych punktów zapisywana jest w plikach tymczasowych. Niestety skorzystanie z nich jest czasochłonne, co wydłużyło czas działania programu. W momencie gdy sąsiednie do aktualnie
przetwarzanego pole przyjmuje wartość K następuje zakończenie wykonywania się zasiedlania labiryntu.

Następnie dochodzi do wywołania funkcji służącej do wyznaczenia od końca trasy prowadzącej przez labirynt. Zaczyna się ona w polu sąsiadującym z końcem labiryntu, i następnie przechodzi do sąsiadującego pola o wartości o 1 mniejszej niż jej własna, aż do napotkania wartości a, z której przechodzi do sąsiadującej wartości z. Funkcja sprawdza kierunek poruszania się w pliku, kiedy dochodzi do jego zmiany zapisuje w pliku
tymczasowym jej kierunek (z perspektywy osoby poruszającej się od początku pliku do jego końca, a więc odwrotnie niż kierunek poruszania się algorytmu) oraz ilość kroków wykonaną na wprost od poprzedniego zakrętu.

Po wypisaniu w ten sposób ścieżki uruchamiana jest funkcja która przepisuje powstały w ten sposób plik tymczasowy do finalnego pliku instructions.txt, odwracając kolejność jego linii.

Te same kroki wykonywane są na pliku binarnym, po przepisaniu go na plik tekstowy.

2.Podział Kodu

Kod podzielony został na moduły:

A)Obsługa Algorytmu BFS (bfs.c, bfs.h)
B)Wykorzystywana w kilku miejscach kodu struktura position, ułatwiająca
znalezienie startu i końca labiryntu, oraz powiązane z jej obsługą funkcje (movement.c, movement.h)
C)Obsługa pliku binarnego, przepisanie na plik tekstowy (directions.c, direcionts.h)
D)Implementacja kolejki przy pomocy plików tymczasowych (queue.c, queue.h)

Testy

1.Labirynt 512x512
Czas: ok. 40.5 sekundy
Pamięć: heap peak = 36576
2.Labirynt 512x256
Czas: ok. 32,5 sekundy
Pamięć: heap peak = 36576
3.Labirynt 7x7
Czas: ok. 0.01 sekundy
Pamięć: heap peak = 36576

Widoczny jest w nich wyraźny wzrost czasu wykonywania funkcji, nie ma jednak istotnego wzrostu zużytej przez program pamięci.

Wnioski

Konieczność optymalizacji wykorzystania pamięci była nowym wyzwaniem, początkowe podejście oparte na standardowej strukturze Queue() zachowywanej w pamięci okazało się zużywać znacznie więcej zasobów niż dostępny nam limit już przy niezbyt dużych labiryntach (512x256). Konieczne było znalezienie lepszej pod tym względem metody. Przy przyszłych projektach ważne jednak byłoby opracowanie metody bardziej opłacalnej pod względem czasu wykonania niż aktualna implementacja plików tymczasowych. Wykonanie projektu pozwoliło nam na nowe doświadczenie poszukiwania bilansu między optymalizacją czasu wykonywania się programu, przy zachowaniu rygoru dostępnej pamięci, oraz przynajmniej częściowe doświadczenie potencjalnych ograniczeń systemowych które mogą przytrafić nam się przy przyszłych projektach na studiach lub w życiu zawodowym. Projekt stanowił dobry sposób odświeżenia, a także poszerzenia naszej wiedzy o języku C.