|  |  |
| --- | --- |
| **JIMP1** | **Inf. Stos. 2023Z** |
| *Mrówka Langtona* | 18.01.2024 |
| Oliwia Pawelec, Jakub Żebrowski G1 | |

Spis treści

[Problem 1](#_Toc156511623)

[Struktura programu 1](#_Toc156511624)

[Wywołanie programu 2](#_Toc156511625)

[Przykładowe działanie programu 3](#_Toc156511626)

[Wnioski 5](#_Toc156511627)

# Problem

Mrówka Langtona to prosty automat komórkowy opracowany przez Chrisa Langtona w 1986. Opiera się na następujących zasadach:

*Mrówka chodzi po dwuwymiarowej siatce. Może poruszać się w jednym z 4-ech kierunków (góra, dół, lewo, prawo), zgodnie z następującymi zasadami:*

*• Mrówka znajduje się w komórce białej, wykonuje: obrót o 90 stopni w prawo, zmienia kolor komórki na przeciwny, przesuwa się o jedną komórkę do przodu*

*• Mrówka znajduje się w komórce czarnej, wykonuje: obrót o 90 stopni w lewo, zmienia kolor komórki na przeciwny, przesuwa się o jedną komórkę do przodu*

Celem naszego projektu było napisane programu, który symuluje ten automat: pokazuje kolejne kroki mrówki na planszach. Każdą planszę (krok) należało wypisać na ekran konsoli lub do wskazanego pliku.

Nasz program prezentuje następujące funkcjonalności:

* Tworzy symulację działania automatu dla i-kroków (liczba podana przez użytkownika)
* Generuje planszę z pustymi polami, a następnie generuje przeszkody (czarne pola) które zajmują łącznie jakiś procent planszy – podany przez użytkownika
* Zapisuje kolejne kroki symulacji do plików z początkową nazwą podaną przez użytkownika – kolejne będą się nazywać *nazwa\_nriteracji* – lub na konsolę jeśli żadna taka nazwa nie została podana
* Umożliwia wczytanie dowolnego stanu planszy do programu i dalsze działanie na nim – użytkownik podaje nazwę pliku, gdzie taka plansza się znajduje

# Struktura programu

Program składa się z 4 plików \*.c i 5 plików nagłówkowych.

Struktura:

* constants.h
* field.c field.h
* ant.c ant.h
* manager.c manager.h
* main.c

Struktura zależności:

Opis poszczególnych części (co robią, jak to robią):

* **constants.h:**

Plik constants.h zawiera stałe zdefiniowane dla programu – znaki specjalne używane do wypisywania planszy i mrówki do plików oraz maksymalne wielkości planszy – do 1000.

* **field.h field.c:**

Pliki field.h i field.c zawierają informację o strukturze planszy, przez którą będzie przechodzić mrówka. Składa się z białych i czarnych pól (wartości 0 i 1) – jest to tablica 2-D field.

typedef struct {

int m, n;

char field[M\_MAX][N\_MAX];

} field\_t;

W tym pliku znajduje się również generator losowych przeszkód (czarnych pól).

void random\_obstacles(field\_t\* field, int pc) {

int blackies = ((field->m \* field->n) / 100) \* pc;

while ( --blackies >= 0 ) {

int x = rand() % field->m;

int y = rand() % field->n;

if ( field->field[x][y] == 0)

field->field[x][y] = 1; // = "█";

else

++blackies;

}

}

Ustala ile łącznie powinno być przeszkód na planszy (blackies), a następnie wyszukuje na planszy pustych komórek, do których może być wstawiona przeszkoda.

* **ant.c ant.h**

Plik nagłówkowy zawiera podstawowe informacje, które każda instancja mrówki będzie posiadać: jej położenie na planszy (x, y) oraz kierunek określony liczbami od 0 do 3, które można rozumieć jako kierunki, w kolejności zgodnie ze wskazówkami zegara: góra, prawo, dół, lewo:

typedef struct {

int x, y; // pozycja mruwy

char direction; // kierunek mruwy

} ant\_t;

Funkcja move\_ant odpowiada za przemieszczanie się mrówki podczas kroku.

Najpierw zmienia kolor komórki, w której się znajduje, na przeciwny, następnie wykonuje krok do przodu – następuje zmiana koordynatów x, y w zależności w który kierunek ma mrówka (ant->direction).

Następnie ustalany jest nowy kierunek mrówki – następuje rotacja, jej kierunek jest uzależniony od koloru komórki w której mrówka znajduje się po przejściu.

void move\_ant(ant\_t\* ant, field\_t\* field) {

// zmień kolor na przeciwny

field->field[ant->x][ant->y] = field->field[ant->x][ant->y] == 0 ? 1 : 0;

// poprzednie koordynaty

int p\_x = ant->x, p\_y = ant->y;

// przesuń mruwe do przodu

switch (ant->direction) {

case 0: // czyli ^

ant->x--;

break;

case 1: // czyli >

ant->y++;

break;

case 2: // czyli v

ant->x++;

break;

case 3: // czyli <

ant->y--;

break;

default:

break;

}

// reset pozycji - czyli gdy mrówka teoretycznie wyjdzie poza planszę - nie powinno być no możliwe!!!!

if ( ant->x < 0 || ant->x >= field->m )

ant->x = p\_x;

if ( ant->y < 0 || ant->y >= field->m )

ant->y = p\_y;

if (field->field[ant->x][ant->y] == 0) { // biały

// w prawo

ant->direction++;

} else { // czarny

// w lewo

ant->direction--;

}

ant->direction = (ant->direction + 4) % 4;

}

* **manager.c manager.h**

Funkcja pleasant\_character zamienia format przechowywania planszy, mrówki w programie na znaki specjalne (widoczne w pliku constants.h).

char\* pleasant\_character(field\_t\* field, ant\_t\* ant, int x, int y) {

int is\_black = (field->field[x][y] == 1);

// na tym polu nie stoi mruwa - zwroć normalny kloc

if (x != ant->x || y != ant->y)

return is\_black == 0 ? BLACK : WHITE;

// ładna mruwka na planszy

if ( ant->direction == 0) // ^

return is\_black == 0 ? BLACK\_UP : WHITE\_UP;

if ( ant->direction == 1) // >

return is\_black == 0 ? BLACK\_RIGHT : WHITE\_RIGHT;

if ( ant->direction == 2) // v

return is\_black == 0 ? BLACK\_DOWN : WHITE\_DOWN;

if ( ant->direction == 3) // <

return is\_black == 0 ? BLACK\_LEFT : WHITE\_LEFT;

return " ";

}

Funkcja get\_from\_file (I jej pochodne, jak read\_input, ant\_strlen, anttaker, antmaker, field\_editor) odpowiadają za pozyskanie stanu planszy z pliku i ściągnięcie go do programu.

// pobieranie stanu planszy ze wskazanego pliku

void get\_from\_file(field\_t\* dest\_field, ant\_t\* dest\_ant, char\* filename) {

FILE \*opener = fopen(filename, "r");

if ( opener == NULL ) {

fprintf(stderr, "Podano nieprawilowy plik wejsciowy. Ups!\n" );

exit(EXIT\_FAILURE);

}

int new\_m = 0, new\_n = 0;

//int ant\_x = 0, ant\_y = 0;

unsigned int utemp = 0;

char\* input\_buf;

while ((input\_buf = read\_input(opener)) != NULL) {

// printf("%04u: %s\n", ant\_strlen(input\_buf), input\_buf);

++new\_m;

if(new\_n == 0) new\_n = ant\_strlen(input\_buf);

if((utemp = anttaker(input\_buf).b) != -1) {

antmaker(dest\_ant, anttaker(input\_buf), input\_buf, new\_m - 1);

}

field\_editor(dest\_field, input\_buf, new\_m - 1);

}

dest\_field->m = new\_m;

dest\_field->n = new\_n;

fclose(opener);

}

* **main.c**

while ((c = getopt(argc, argv, "m:n:i:d:s:o:p:")) != -1) {

switch (c) {

case 'm': {

m = atoi(optarg);

break;

}

case 'n': {

n = atoi(optarg);

break;

}

case 'i': {

it\_count = atoi(optarg);

break;

}

case 'p': {

p = atoi(optarg);

break;

}

case 'd': {

p = atoi(optarg);

break;

}

case 's': {

file\_saved = 1;

save\_filename = malloc(strlen(optarg));

memcpy(save\_filename, optarg, strlen(optarg));

save\_filename[strlen(optarg)] = '\0';

printf("Zapisac w = %s\n", optarg);

break;

}

case 'o': {

// jesli o, to tylko z i, ewentualnie z s!

file\_opened = 1;

open\_filename = malloc(strlen(optarg));

memcpy(open\_filename, optarg, strlen(optarg));

open\_filename[strlen(optarg)] = '\0';

printf("Otworzyc z = %s\n", optarg);

break;

}

case '?': {

printf("Nieznany argument: %c\n", optopt);

help(argv[0]);

break;

}

default: { // używanie

help(argv[0]);

exit(EXIT\_FAILURE);

}

}

}

Funkcja getopt pozwala na obsługę argumentów. Dla każdego przypadku (poza -o i -s, których spodziewanym argumentem jest char\*) zamieniamy przyznany argument na liczbę całkowitą, korzystając z funkcji atoi.

Obsługa błędów jest przeprowadzona niżej:

// wykrywanie błędów (dla numeru iteracji)

if ( it\_count < 1 || it\_count > 2000) {

printf("Niepoprawna liczba iteracji (i): %i\n", it\_count);

help(argv[0]);

exit(EXIT\_FAILURE);

}

// wykrywanie błędów (dla nie-otwierania pliku)

if ( m < 5 || m > 1000) {

printf("Niepoprawna liczba kolumn (m): %i\n", m);

help(argv[0]);

exit(EXIT\_FAILURE);

}

if ( n < 5 || n > 1000) {

printf("Niepoprawna liczba wierszy (n): %i\n", n);

help(argv[0]);

exit(EXIT\_FAILURE);

}

if ( p < 0 || p > 100) {

printf("Niepoprawny %% zapelnienia (p): %i\n", p);

help(argv[0]);

exit(EXIT\_FAILURE);

}

if ( d < 0 || d > 3) {

printf("Niepoprawny kierunek mruwki: %i\n", d);

help(argv[0]);

exit(EXIT\_FAILURE);

}

Rozmiary planszy (m, n) muszą należeć do przedziału <5; 1000>, procent wypełnienia do <0; 100>, a kierunek – wspomniany wcześniej – <0; 3>

# Wywołanie programu

Program można skompilować za pomocą komendy make:

make

Lub ręcznie skompilować:

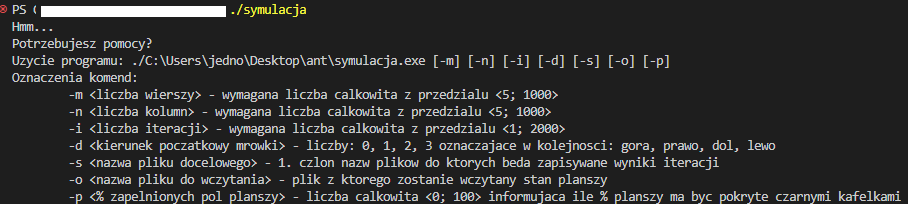
gcc -Wall field.c ant.c manager.c main.c -o mrowka

Następnie wywołać za pomocą:

./symulacja [komendy]

Aby wywołać komendę help (komendę z poleceniami) należy wpisać program bez argumentów:

./symulacja



Rys. 1 Działanie wywołania ./symulacja (plik wykonywalny programu)

Program można użyć, wpisując następujący format do konsoli:

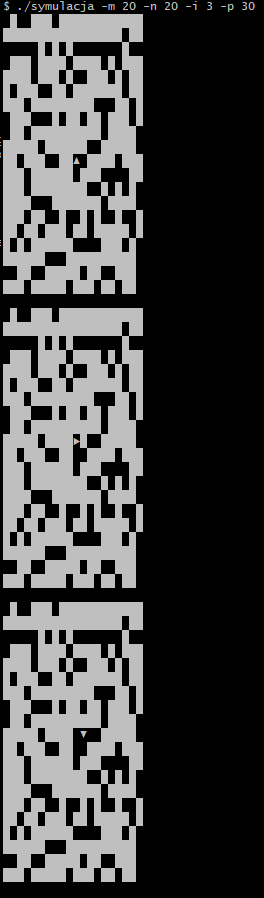
./symulacja [-m] [-n] [-i] [-d] [-s] [-o] [-p]

Gdzie:

* m <liczba wierszy> - wymagana liczba całkowita z przedziału <5; 1000>
* n <liczba kolumn> - wymagana liczba całkowita z przedziału <5; 1000>
* i <liczba iteracji> - wymagana liczba całkowita z przedziału <1; 2000>
* d <kierunek początkowy mrówki> - liczby: 0, 1, 2, 3 oznaczające w kolejności: góra, prawo, dół, lewo
* s <nazwa pliku docelowego> - 1. człon nazw plików do których Beda zapisywane wyniki iteracji
* o <nazwa pliku do wczytania> - plik z którego zostanie wczytany stan planszy
* p <% zapełnionych poł planszy> - liczba całkowita <0; 100> informująca ile % planszy ma być pokryte czarnymi kafelkami

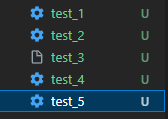
# Przykładowe działanie programu

* m=20 n=20 i=3 %=30

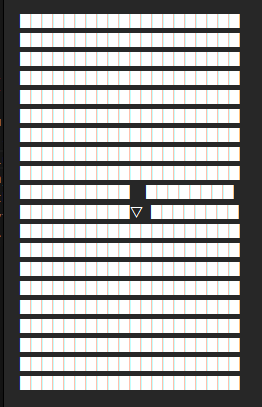


Rys.

* m=20 m=20 i=5 s=test

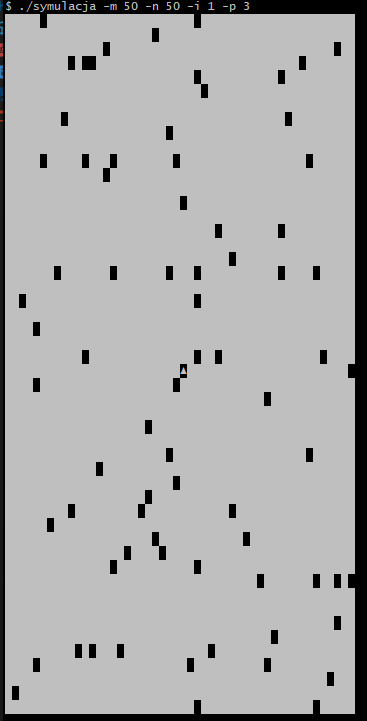


Rys.



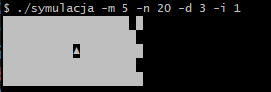
Rys. Plik test\_5

* m=50 n=50 i=1 p=3



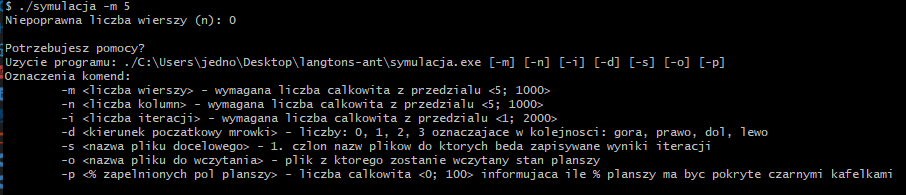
Rys.

* m=5 n =20 d=3 i=1



Rys.

* bez podania liczby wierszy



Rys.

# Wnioski

Problem automatu nie był sam w sobie trudny – jego algorytm jest trywialny, a sama implementacja chodzącej mrówki była przyjemna (trójkąt-mrówka zmieniająca swoje pozycje między plikami wyglądała doprawdy przezabawnie).

Największym wyzwaniem było ~~pisanie tego sprawozdania~~, wprowadzenie funkcji getopt do programu – pierwszy raz mieliśmy z tym styczność – testowanie kodu, używanie gita oraz koordynacja pracą grupową.