Specyfikacja implementacyjna automat komórkowy **"The game of life"**

Jan Starczewski
Bartosz Michałowski

15 marca 2018

Spis treści

1 Informacje ogólne

1.1 Nazwa programu

Nazwa programu: `life_game_generator`

1.2 Język

Program został napisany w języku C i jest przystosowany do uruchomienia w standardzie znajdującym się na serwerze ssh: *ssh.jimp.iem.pw.edu.pl*.

1.3 Uruchomienie programu

Program jest przeznaczony do uruchomienia z interfejsu tekstowego. Odpowiednie parametry przekazywane w linii poleceń pozwalają sterować programem i uzyskać oczekiwane rezultaty.

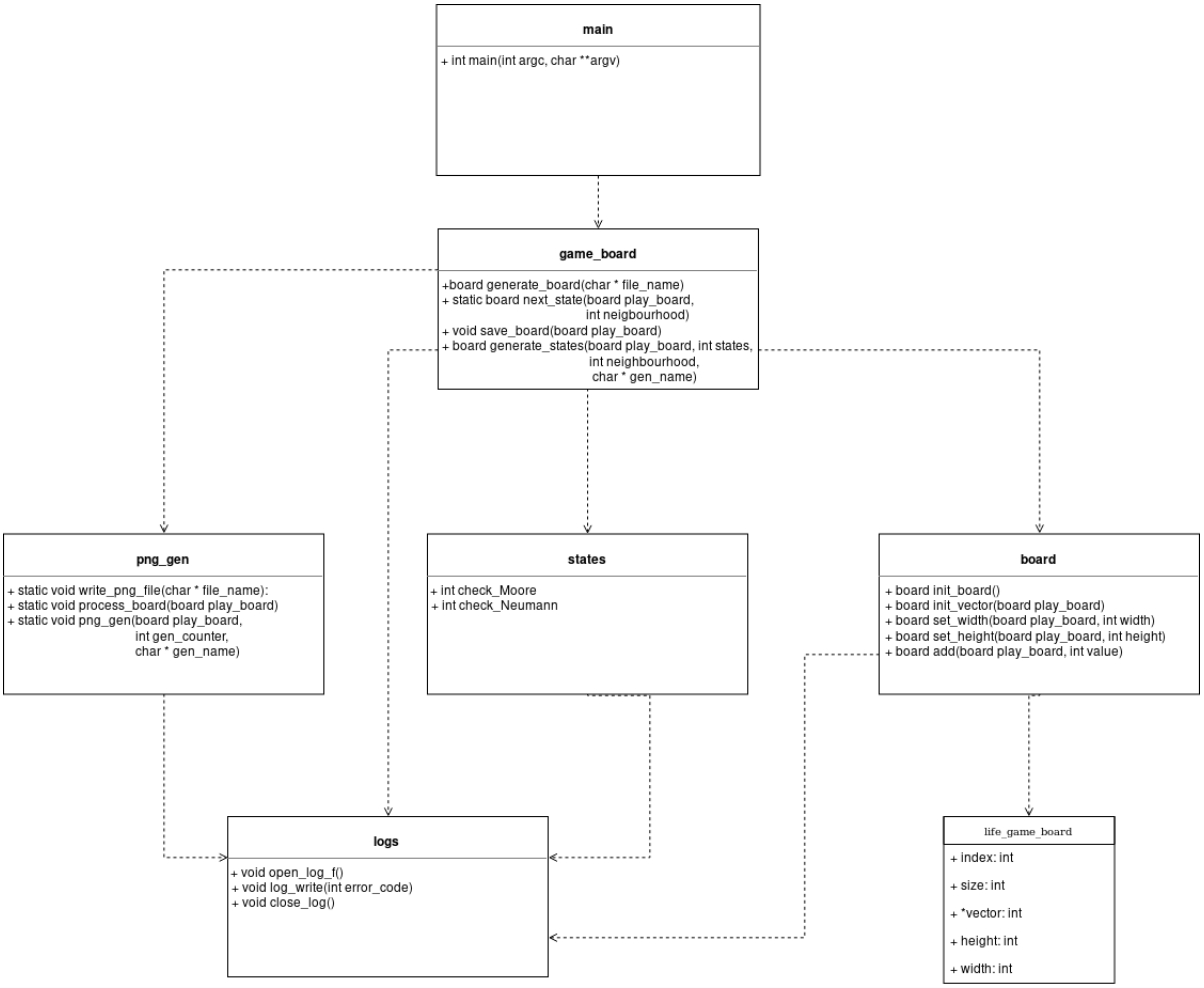
1.3.1 Wywołanie

`life_game_gen [ścieżka generacji początkowej] [liczba generacji] [rodzaj sąsiedztwa] [nazwa katalogu wyjściowego]`.

1.3.2 Przykładowe wywołanie

```
life_game_gen fgen.txt 10 -M 10gen
```

2 Diagram modułów



Rysunek 1: Diagram modułów

2.1 Relacje modułów

Moduł `main` zawiera funkcję sterującą, komunikuje się on bezpośrednio z modułem `game_board`, który jest odpowiedzialny za obsługę planszy i przeprowadzanie kolejnych generacji. Moduł ten jest bezpośrednio związany z modułem `board`, w którym znajduje się struktura (oraz funkcje obsługujące ją), na której operuje. Dodatkowo moduł `game_board` jest związany z modułem `png_gen` odpowiedzialnym za generację plików graficznych i modułem `states`, w którym znajduje się algorytm. Wszystkie modułu pośrednio, lub bezpośrednio są powiązane z modułem `log` odpowiedzialnym

za generowanie pliku tekstowego z wiadomościami do użytkownika końcowego.

3 Opis modułów

3.1 Moduł `main`

Moduł main składa się jedynie z funkcji domyślnej `int main(int argc, char **argv)` przyjmującej argumenty oraz ich ilość przy wywołaniu ze środowiska tekstowego. Jej zadaniem jest interpretacja argumentów wejściowych programu i przekazanie odpowiednich wartości do dalszych modułów w celu przeprowadzenia generacji. W wypadku nieodpowiedniego formatu argumentów odpowiednie informacje przekazywane są do modułu log.

3.2 Moduł `game_board`

Moduł `game_board` komunikuje się z funkcją sterującą. Składowe modułu odpowiadają za poprawną alokację pamięci dla potrzebnych struktur danych, inicjację procesu generacji komórek, wytwarzanie nowych pokoleń i wszczęcie procedury generacji plików graficznych. W module znajdują się tez funkcje umożliwiające wydrukowanie stanu komórek, służy to testowaniu ręcznemu obranemu jako ogólny sposób testowania modułów.

3.2.1 Funkcje wewnątrz modułu

1. Funkcja `board generate_board(char* filename)` przyjmuje jako argument nazwę pliku, w którym znajduje się konfiguracja początkowa komórek, analizuje plik i czyta zawarte w nim wartości, zwraca wskaźnik na strukturę, wewnątrz której jest wektor reprezentujący przeczytaną z pliku konfigurację, wymiary planszy, parametry.

2. Funkcja `static board next_state(board play_board, int neighbourhood)` otrzymuje wskaźnik na strukturę, która przechowuje stan komórek oraz liczbę definiującą zasady (sąsiedztwo) na podstawie, których będą generowane kolejne pokolenia. Funkcja odwołuje się do algorytmu definiującego stan kolejny i zwraca wskaźnik na strukturę zawierająca konfigurację nowego pokolenia. Słowo `static` poprzedza w deklaracji typ zwracany, ponieważ algorytm nie powinien być dostępny z zewnątrz pliku, bez zachowania odpowiednich procedur. Przed zwróceniem wskaźnika, zwalniana jest pamięć zarezerwowana dla poprzedniej generacji.

3. Funkcja `board generate_states(board play_board, int states, const char* neighbourhood)` przyjmuje wskaźnik na strukturę przechowującą stan komórek, liczbę reprezentującą ilość plików PNG do

wygenerowania oraz wartość liczbową odpowiadającą zasadom, na podstawie których będą generowane kolejne pokolenia. Wewnątrz następuje wywołanie funkcji generujących plik PNG i kolejną generację komórek. Zwracany jest wskaźnik na strukturę zawierająca konfigurację końcową.

4. Funkcja `void prin_board(board play_board)` otrzymuje wskaźnik na strukturę zawierającą ułożenie komórek i wypisuje je w formacie tekstowym na `stdout` zachowując odpowiednią szerokość i wysokość. Funkcja jest głównie używana do testowania poprawności działania wyżej opisanych składowych modułu.

3.3 Moduł `board`

Moduł `board` przechowuje strukturę danych odpowiadającą planszy. W skład struktury wchodzi jednowymiarowy wektor o długości równej iloczynowi szerokości i wysokości, pola reprezentujące wymiary planszy i pola do obsługi wektora. Wewnątrz modułu występują funkcje odpowiadające za inicjację wektora, przypisanie wartości do pól, czy dodanie do niego wartości.

3.3.1 Struktura danych imitująca planszę

Struktura danych imitująca planszę jest zadeklarowana w sposób umożliwiający przekazywanie wszystkich danych na temat planszy z przekazaniem wskaźnika na ową strukturę. Przyjmuje na postać:

```
typdef struct life_game_board {
   int index;
   int size;
   int *vector;
   int height;
   int width;
} *board;
```

3.3.2 Funkcje wewnątrz modułu

1. Funkcja `board init_board()` alokuje pamięć dla struktury przechowującej wszystkie dane planszy, przypisuje wartość początkową dla pola `size` i zwraca wskaźnik na strukturę `life_game_board`

2. Funkcja `board init_vector(board play_board)` przyjmuje jako argument wskaźnik na strukturę danych planszy. W ciele następuje przypisanie wartości pola `size` i alokacja pamięci dla wektora przechowującego stany komórek. Zwracaną wartością jest argument przekazany do funkcji.

3. Funkcja `set_width(board play_board, int width)` otrzymuje wskaźnik na strukturę przechowującą dane planszy i przypisuje polu `width` wartość reprezentowaną przez drugi argumentu definiując tym szerokość planszy. Zwraca wskaźnik na strukturę przekazaną do funkcji w pierwszym argumencie.

4. Funkcja `set_height(board play_board, int height)` otrzymuje wskaźnik na strukturę przechowującą dane planszy i przypisuje polu `height` wartość reprezentowaną przez drugi argument, definiując tym wysokość planszy. Zwraca wskaźnik na strukturę przekazaną do funkcji w pierwszym argumencie.

5. Funkcja `add(board play_board, int value)` otrzymuje wskaźnik na strukturę przechowującą dane planszy i przypisuje polu w wektorze `vector[board->index]` wartość reprezentowaną przez drugi argument, instrukcja ta dodaje komórkę w odpowiednim stanie do wektora. Funkcja zwraca wskaźnik na strukturę przekazaną jaki pierwszy argument.

3.4 Moduł `png_gen`

Moduł `png_gen` odpowiada za generację wynikowych plików png. Korzysta on z biblioteki `png.h`, która pozwala na obsługę takich plików. Jest to moduł zbudowany na wzór tego, dostępnego na serwisie ISOD. Wprowadzona modyfikacja pozwoli na skalowanie grafiki dzięki czemu odpowiednie pola są większe i możliwe do dostrzeżenia bez potrzeby przybliżania. Całością kieruje jedna funkcja sterująca.

3.4.1 Funkcje wewnątrz modułu

1. Funkcja `png_gen` jest funkcją sterującą modułu. Przyjmuje wskaźnik na strukturę planszy i numer generacji a następnie przekazuje ją do kolejnych funkcji. Odpowiada też za utworzenie nazwy pliku wyjściowego.

2. Funkcja `process_board` przetwarza planszy z postaci binarnej do formatu png, możliwego do zapisania w pliku. Jako argument przyjmuje wskaźnik na planszę. W ciele jej zawarta jest też instrukcja sterująca skalowaniem planszy.

3. Funkcja `write_png_file` zapisuje przekonwertowaną planszę do pliku wynikowego w folderze `./output/nazwa_generacji`. Otrzymuje tablicę znaków, która jest nazwą pliku wynikowego.

3.5 Moduł `log`

Moduł `log` odpowiada za utworzenie pliku log w katalogu głównym programu i zapis do niego odpowiednich komunikatów. Zawiera trzy funkcje odpowiadające za utworzenie pliku, zapis komunikatów i zamknięcie pliku po zakończeniu działania programu.

3.5.1 Funkcje wewnątrz modułu

1. Funkcja `void open_log()` tworzy plik log i udostępnia go do użytku dla pozostałych funkcji modułu. W wypadku niepowodzenia wypisuje stosowny komunikat korzystając z wyjścia standardowego. Nie przyjmuje żadnych argumentów.

2. Funkcja `void write_message(int log_num)` zapisuje odpowiednie komunikaty do utworzonego wcześniej pliku log. Przyjmuje liczbę całkowitą która jest numerem komunikatu, który należy wypisać i których lista znajduje się wewnątrz funkcji.

3. Funkcja `void close_log()` odpowiada za zamknięcie pliku log tuż przed zakończeniu działania programu. Nie przyjmuje żadnych argumentów.

3.6 Moduł `states`

Moduł `states` zawiera w sobie algorytm, który sprawdza stan komórek dookoła rozpatrywanej. Dla zachowania przejrzystości algorytmu, co przekłada się na łatwość jego zrozumienia (dla osób niezaznajomionych z kodem), został on rozdzielony w zależności od sąsiedztwa(zasad) na podstawie, których powstają generację. Poskutkowało to duplikacjami w kodzie.

3.6.1 Funkcje wewnątrz modułu

1. Funkcja `int check_Moore(board play_board, int index)` przyjmuje wskaźnik na strukturę przechowującą komórki i indeks rozpatrywanej komórki. Funkcja rozpatruje sąsiedztwo Moore, czyli sprawdza osiem komórek. W ciele jej istnieje pole typu całkowitego `neighbours_state`, z przypisaną wartością początkową równą zero. Reprezentuje ono ilość żywych komórek wokół rozpatrywanej. Pole to ulega inkrementacji gdy jedna ze sprawdzanych komórek jest żywa. Na koniec funkcja zwraca wartość pola `neighbours_state`.

2. Funkcja `int check_Neumman(board play_board, int index)` przyjmuje wskaźnik na strukturę przechowującą komórki i indeks rozpatrywanej komórki. Funkcja rozpatruje sąsiedztwo von Neummana, czyli sprawdza cztery komórki. W ciele jej istnieje pole typu całkowitego.

`neighbours_state`, z przypisaną wartością początkową równą zero. Reprezentuje ono ilość żywych komórek wokół rozpatrywanej. Pole to ulega inkrementacji gdy jedna ze sprawdzanych komórek jest żywa. Na koniec funkcja zwraca wartość pola `neighbours_state`.

4 Algorytm sprawdzający i przechowywanie danych w programie

Wcześniejsza znajomość zagadnienia, mianowicie skończona ilość sąsiedztw, kształt planszy i zasady poruszania się po niej, przełożyły się na dogłębne zrozumienie problemu. **Przyjęto konwencję przechodzenia komórek, po dojściu do krańca wygenerowanej planszy, pojawiają się one po przeciwległej stronie**. Napisany został algorytm, o niskiej złożoności obliczeniowej, co przekłada się na szybkość wykonywania. Sposób implementacji pozwala na szybkie zrozumienie jego sposobu działania i łatwe testowania. Struktura danych przechowująca wszystkie informacje potrzebne do przeprowadzania kolejnych generacji, również została stworzona z uwzględnieniem założeń na jakich powstał wyżej wymieniony algorytm. Postawiono na prostotę implementacji i obsługi owej struktury. Wszystkie dane przechowywane są w postaci pól typu całkowitego i jednego liniowego wektora przechowującego stan komórek.

4.1 Algorytm sprawdzający

Na potencjalną niską złożoność obliczeniową, szybkość wykonywania i prostotę wpływa:

1. Operowanie na jednej konkretnej strukturze danych.
2. Brak iteracji po planszy komórek wewnątrz algorytmu.
3. Każda komórka będąca sąsiadem rozpatrywanej jest osiągana za pomocą odwołania się do jej indeksu.
4. Intuicyjne blokowe sterowanie pozwala na wyobrażenie procesu sprawdzania stanu komórek sąsiadujących.

4.2 Przechowywanie danych w programie

Zastosowanie jednowymiarowego wektora jest niezbędne do zachowania szybkości wyżej opisanego algorytmu. Główne zalety przechowywania wszystkich danych potrzebnych do przeprowadzenia generacji w jednej strukturze, w której prócz wektora znajdują się tylko pola typu całkowitego to:

1. Łatwość deklaracji i inicjalizacji wszystkich danych potrzebnych do przeprowadzania generacji (brak inicjalizowania mniejszych struktur w pętli).

2. Prostota w zwalnianiu pamięci nieużywanych już struktur danych.

3. Większa kontrola nad kodem, łatwość naprawy błędów, z powodu prostoty modułu odpowiedzialnego za obsługę struktury.

5 Wymagania dotyczące użytkowania programu

5.1 Kompilacja

Program należy skompilować korzystając zapewnionego pliku Makefile, który zawiera odpowiednie polecenia dla kompilatora. Kompilacja przeprowadzana jest poleceniem make wykonanym w folderze zawierającym pliki programu. Wymagany jest kompilator obsługujący standard języka C89 lub wyższy. Utworzony w wyniku kompilacji plik `life_game_gen` jest plikiem wykonywalnym. Dla poprawnej kompilacji niezbędne jest zapewnienie dostępu do odpowiednich bibliotek.

5.2 Biblioteki

Program w znacznym stopniu korzysta z bibliotek zawartych w bibliotece standardowej języka C, takich jak `string.h, stdio.h` bądź `stdlib.h`. Jednakże do umożliwienia utworzenia plików o formacie png wymagana jest zewnętrzna biblioteka `lib_png` w wersji 1.5.0 lub wyższej, dostępna na stronie `www.libpng.org`. Jednocześnie wymaga ona biblioteki zlib w wersji 1.2.5 lub wyższej, dostępnej na stronie `zlib.net`.

5.3 Obciążenie systemu przez program

Podczas działania algorytmu obciążenie jest minimalne. W pamięci przechowywana jest jedynie tablica liczb całkowitych opisana wcześniej w module board. W trakcie zapisu do plików png obciążenie nieznacznie wzrasta, ze względu na charakter tego formatu, jednakże zapis odbywa się wierszami, co usprawnia jego działanie i ogranicza możliwość wystąpienia deficytu dostępnej pamięci operacyjnej. Ilość zajmowanych zasobów maszyny w trakcie działania programu, jak i rozmiar wygenerowanych plików wzrastają wraz z rozmiarem obsługiwanej planszy dostarczonej przez użytkownika.

6 Testowanie

6.1 Konwencja

Testowanie wodzące w projekcie to testowanie regresyjne. Ręczne sprawdzenie każdej powstałej funkcjonalności za pomocą odpowiednich funkcji wewnątrz modułów jest wystarczające dla osiągnięcia założonych celów odnośnie testowania funkcjonalności. Przykładową funkcją, która jest zaangażowana w proces testowania jest `void print_board(board play_board)` z modułu `game_board`. Odpowiada ona za wypisanie planszy komórek. Reszta funkcji będzie dopisywana w samym procesie tworzenia kodu, gdyż ich prostota względem całego projekut nie wymaga opisu w sekcji *Opis modułów*.

6.2 Użyte narzędzia

Prócz testowania regresyjnego, zostaną użyte poniższe narzędzia:

1. Program *Valgrind* do debugowania pamięci na platformie Linux
2. Program *Dr. Memory* do debugowania pamięci na platformie Windows
3. Składowe zintegrowanego środowiska deweloperskiego *CLlion*, mianowicie
   (a) *CLion Debuger* do sprawdzania miejsc wystąpień wycieków pamięci.
   (b) *CLion Analitics* do sprawdzenia, czy ogól dobrych praktyk odnośnie implementacji kodu został zachowany.

6.3 System

Program będzie testowany i uruchamiany na poniższych systemach i kompilatorach

1. Windows 8.1, kompilator MinGW
2. Fedora, GCC
3. Kompilator znajdujący się na serwerze ssh: *ssh.jimp.iem.pw.edu.pl*

7 Wersjonowanie

Wersjonowanie projektu jest oparte o system kontroli wersji *Git*. Wersje programu są przechowywane w repozytorium `2018_JIMP2_repozytorium_gr1` stworzonym na potrzeby projektu. Kolejne wersje umieszczono w gałęzi `master` wyżej wymienionego repozytorium.

8 Narzędzia

Narzędzia użyte w procesie tworzenia programu to:

1. Zintegrowane środowisko deweloperskie *CLion*
2. Edytor tekstu *VIM*
3. System kontroli wersji *Git*
4. Debugger pamięci *Valgrind*
5. Debugger pamięci *Dr. Memory*
6. Klieny serwera SSH *Putty*