Praca domowa I—Automaty komórkowe

12 listopada 2026

1 Instrukcja

Napisz framework do symulacji automatów komórkowych wraz z aplikacją terminalową, która go uruchamia. Automat komórkowy to siatka komórek. W każdym kroku automatu każda komórka oblicza swój nowy stan na podstawie własnego stanu oraz stanów swoich sąsiadów, po czym automat jednocześnie aktualizuje stan wszystkich komórek. Najbardziej znanym przykładem jest Gra w życie Conwaya. Celem projektu jest stworzenie frameworka do symulacji automatów komórkowych, który powinien być generyczny ze względu na typ komórki i regułę, która nią rządzi.

Termin projektu: 3.12.2026. Każdy tydzień opóźnienia skutkuje ograniczeniem maksymalnej liczby punktów o 5.

W sekcji 2 znajdziesz wymaganą do zaimplementowania funkcjonalność. Za jej pełną implementację otrzymać możesz 10pkt. W sekcji 3 znajdziesz dodatkową funkcjonalność. Przy każdej funkcjonalności znajduje się liczba dodatkowych punktów za jej implementację i jej identyfikator. Aby otrzymać punkty za dodatkową funkcjonalność, musisz zaimplementować funkcjonalność z sekcji 2 za przynajmniej 8 punktów. W sekcji 4 znajdziesz rzeczy, których nie możesz robić w projekcie.

Jeśli w poleceniu jest napisane, że należy zaimplementować strukturę/cechę/funkcję o nazwie `X`, która nie jest jednak definiowana szczegółowo przez polecenie, to jej implementacja jest dowolna.

Całkowicie zabronione jest używanie bibliotek innych niż te, które dozwolone są w poleceniu. Projekty używające innych bibliotek nie będą sprawdzane.

Dozwolone jest używanie dowolnych elementów z biblioteki standardowej, o ile nie są one zakazane przez sekcję 4.

Ocena za projekt liczona jest ze wzoru

$$k = \min(r + a + p, 15),$$

gdzie k—ocena końcowa, r—punkty z sekcji 2, $a = 0$ jeśli $r < 8$, lub a jest równe punkty z sekcji 3 jeśli $r \geq 8$, p—punkty z sekcji 4 (zero lub liczba ujemna).

2 Wymagana funkcjonalność

1. (2pkt) Podstawowe cechy.
   (a) Cecha `Cell` opisująca stan pojedynczej komórki automatu (niezwiązana z `std::cell::Cell`). Stan komórki można klonować i porównywać oraz ma on wartość domyślną—stan „pusty”, używany dla wszystkich komórek znajdujących się poza siatką. Stan ten można zamienić na znak używany w plikach tekstowych i przy wypisywaniu oraz odwrotnie.
   (b) Enum `Neighbourhood` z wariantami `Moore` i `VonNeumann` (sąsiedztwo Moore'a i von Neumanna), który potrafi wypisać pozycje sąsiadów zadanej komórki względem niej.
   (c) Cecha `Rule<C: Cell>` opisująca, jak zmieniają się komórki. Na podstawie aktualnego stanu komórki i stanów jej sąsiadów zwraca następny stan tej komórki. Wskazuje także, z którego `Neighbourhood` korzysta. Kolejność, w jakiej sąsiedzi przekazywani są do reguły, jest dowolna, ale musi być udokumentowana.

2. (2pkt) Siatka.

   (a) Generyczna struktura `Grid<C: Cell>` o szerokości i wysokości ustalanych w czasie działania programu oraz enum `EdgeMode` z wariantami `Dead` (komórki poza siatką mają wartość `C::default()`) i `Wrap` (siatka jest torusem).

   (b) Tworzenie siatki, odczyt komórki i zmiana komórki. Użycie współrzędnych spoza siatki zwraca błąd.

   (c) Obliczanie jednego pokolenia z użyciem dowolnej `Rule` dla typu komórek siatki. Wszystkie nowe stany muszą być obliczane wyłącznie na podstawie poprzedniego pokolenia.

   (d) `Display` dla siatki, wypisujące jedną linię znaków na wiersz.

   (e) Struktura `Simulation` generyczna ze względu na typ komórki i regułę, zawierająca siatkę, regułę i licznik pokoleń.

3. (2.5pkt) Trzy zestawy reguł. Każdy ma własny typ komórki implementujący `Cell` i własną strukturę implementującą `Rule`.

   (a) (0.75pkt) **Gra w życie**: dwa stany, sąsiedztwo Moore'a. Martwa komórka z dokładnie 3 żywymi sąsiadami ożywa. Żywa komórka z 2 lub 3 żywymi sąsiadami pozostaje żywa. We wszystkich pozostałych przypadkach komórka jest martwa w następnym pokoleniu.

   (b) (0.75pkt) **Brian's Brain**: trzy stany (wyłączona, włączona, umierająca), sąsiedztwo Moore'a. Wyłączona komórka z dokładnie 2 *włączonymi* sąsiadami włącza się; włączona komórka staje się umierającą; umierająca komórka wyłącza się. Umierające komórki nie liczą się jako włączeni sąsiedzi.

   (c) (1pkt) **Mrówka Langtona**: sąsiedztwo von Neumanna. Każda komórka ma kolor i może zawierać mrówkę, zwróconą w jednym z czterech kierunków. W jednym kroku mrówka skręca w prawo na białej komórce i w lewo na czarnej, zmienia kolor komórki i przesuwa się o jedną komórkę do przodu. Ponieważ każda komórka aktualizowana jest niezależnie, mrówki nie przesuwa komórka, na której stoi: zamiast tego komórka przejmuje mrówkę, jeśli mrówka któregoś z jej sąsiadów przesuwa się na nią. Mrówka, która opuści siatkę w trybie `Dead`, znika.

   Do typów komórek używaj enumów; nie reprezentuj stanów za pomocą kilku pól `bool`.

4. (1pkt) Wczytywanie wzorców.

   (a) Parsowanie tekstu do siatki o rozmiarze wynikającym z tekstu: każda linia to wiersz, a każdy znak to komórka.

   (b) Kopiowanie (wklejanie) mniejszej siatki na większą w podanym miejscu.

   (c) Błędy dla: nieznanego symbolu (z numerem linii i kolumny), wierszy o różnych długościach, pustego wzorca oraz wzorca, który nie mieści się w podanym miejscu.

   Przykładowe pliki wzorców dołączone do polecenia używają poniższych symboli, więc Twój parser musi obsługiwać przynajmniej je:

Gra w życie	`.` martwa, `#` żywa
Brian's Brain	`.` wyłączona, `#` włączona, `o` umierająca
Mrówka Langtona	`.` biała, `#` czarna, `^`, `>`, `v`, `<` mrówka na białej komórce zwrócona w górę, w prawo, w dół, w lewo

   Pozostałe stany mrówki Langtona (mrówka na czarnej komórce) mogą używać dowolnie wybranych symboli.

5. (1.5pkt) Aplikacja.

   (a) Podział projektu na bibliotekę (definicja modułów w `lib.rs`) i część użytkową (definicja modułów w `main.rs`). Część użytkowa zawiera wyłącznie parsowanie argumentów, główną pętlę i wypisywanie do terminala; biblioteka zawiera resztę kodu.
   (b) Opcje wiersza poleceń pozwalające wybrać regułę, plik ze wzorcem, rozmiar siatki, tryb krawędzi, liczbę kroków i opóźnienie między klatkami. Możesz użyć biblioteki *clap*.
   (c) Wzorzec umieszczany jest na środku siatki o podanym rozmiarze. Bez pliku ze wzorcem siatka wypełniana jest losowo (możesz użyć biblioteki *rand*).
   (d) W każdej iteracji aplikacja czyści terminal, wypisuje siatkę i numer pokolenia oraz czeka przez podane opóźnienie.
   (e) Błędy z biblioteki wypisywane są w czytelny sposób i skutkują niezerowym kodem wyjścia.

6. (1pkt) Typ `enum Error` i testy.
   (a) Enum zawiera warianty dla wszystkich kategorii błędów, które można napotkać, np. `OutOfBounds`, `UnknownSymbol`, `IoError(std::io::Error)`, itd. Zalecane jest użycie biblioteki *thiserror*. Funkcje w modułach biblioteki powinny przekazywać błędy dalej do kodu aplikacji, chyba że potrafią same sobie z nimi poradzić.
   (b) Testy w każdym napisanym module. Każdy automat, oba tryby krawędzi i każdy rodzaj błędu muszą być przetestowane.

7. Plik `README.md`, który zawiera:
   (a) Opis struktury projektu.
   (b) Opis funkcjonalności nieopisanych w poleceniu.
   (c) Twój ulubiony moduł i uzasadnienie, dlaczego jest ulubiony.

8. Plik `ADDITIONS` zawierający listę zaimplementowanych funkcjonalności z sekcji 3 w postaci listy ich identyfikatorów, np. plik ten może wyglądać tak:

```
rle
tui
```

3 Dodatkowa funkcjonalność

1. (2pkt, `rule_strings`) Reguła `LifeLike` konfigurowana w czasie działania programu napisem w notacji `B/S`, np. `B3/S23` (Life), `B36/S23` (HighLife) lub `B2/S` (Seeds). `B` wymienia liczby żywych sąsiadów, przy których martwa komórka ożywa, `S` wymienia liczby żywych sąsiadów, przy których żywa komórka pozostaje żywa. Regułę można wybrać za pomocą `--rule B36/S23`; niepoprawne napisy są zgłaszane jako błędy.

2. (2pkt, `rle`) Odczyt i zapis wzorców w formacie RLE. Wybierany na podstawie rozszerzenia pliku lub za pomocą opcji.

3. (2pkt, `analysis`) Wykrywanie cykli w ewolucji symulacji: układu stałego lub oscylatora, wraz z jego okresem. Opcja zapisująca liczbę żywych komórek w każdym pokoleniu do pliku CSV.

4. (2pkt, `topology`) Dodatkowa topologia siatki, np. siatka heksagonalna z 6 sąsiadami, wraz z co najmniej jedną regułą, która z niej korzysta. Siatka musi być także wyświetlana w terminalu w sposób możliwie naturalny dla tej topologii (np. dla siatki heksagonalnej: komórki rysowane na dwóch znakach i co drugi wiersz przesunięty o jeden znak). Dodanie jej nie powinno wymagać zmian w już istniejących regułach.

5. (1pkt, `colour`) Stany komórek wyświetlane w terminalu jako symbol wraz z kolorem (np. z użyciem ANSI escape sequences), na przykład różne kolory dla stanów Brian's Brain i dla mrówki. Kolory powinny pochodzić z biblioteki, a nie być zakodowane na stałe w aplikacji.

6. (1pkt, `automaton`) Inny nietrywialny automat dowolnie wybrany przez Ciebie, z co najmniej 4 stanami i regułą, której nie da się wyrazić jako napisu `B/S` w stylu Life, np. Wireworld. Dołącz plik ze wzorcem demonstrującym go i opisz go w `README.md`. Punkty za ten element przyznawane są tylko raz, niezależnie od liczby zaimplementowanych automatów.

7. (3pkt, `tui`) Interfejs TUI zaimplementowany przy pomocy biblioteki *ratatui*, razem z biblioteką *crossterm*. Powinien pozwalać na wstrzymywanie symulacji, wykonywanie pojedynczych kroków, zmianę jej szybkości i przełączanie stanów komórek za pomocą kursora.

8. (1-3pkt, `custom`) Dowolna inna dodatkowa funkcjonalność, oceniana subiektywnie przez prowadzącego (najlepiej skonsultować przed terminem oddania).

4 Praktyki zakazane

Pojawienie się któregokolwiek z poniższych elementów w projekcie skutkuje odjęciem tylu punktów, ile sprecyzowano.

1. Użycie metod `unwrap`, `expect` lub makra `panic`. −1pkt za 1 lub 2 użycia, −2pkt za więcej użyć. Dozwolone jednak używanie w testach.

2. Używanie struktury, gdzie właściwy byłby `enum`, to jest trzymanie w strukturze pól, których wartość czasami ma, a czasami nie ma znaczenia, np.

   ```
   struct IP {
    v4: String,
    v6: String,
    is_v4: bool
   }
   ```

   -1pkt za jedno wystąpienie, -2pkt za więcej.

3. Ignorowanie błędów, kiedy powinny być obsłużone, np. wołanie funkcji, która zwraca `Result<T, E>` i sprawdzanie przez `if let` tylko przypadku `Ok`, kiedy błąd ma znaczenie (jeśli nie ma, to oczywiście dozwolone). -1pkt za jedno lub 2 wystąpienia, -2pkt za więcej.

4. Funkcje dłuższe niż 30 linijek. -1pkt za 1 lub 2, -2pkt za więcej.

5. Funkcje, które niepotrzebnie przejmują własność obiektów, kiedy mogłyby je tylko pożyczać. -1pkt za 1-3 takich funkcji. -2pkt za więcej.

6. Używanie `clone`, kiedy jest niepotrzebny (tzn. kod kompiluje się po jego usunięciu, a nie wpływa to na logikę). -1pkt za jedno lub więcej użyć.

7. Używanie `Rc`, `Arc`, `RefCell`, `Mutex` lub `std::cell::Cell`, -2pkt (własna cecha `Cell` jest oczywiście dozwolona).

8. Używanie `unsafe`, -2pkt.