**Symulacja Pożaru Lasu**

**Radosław i Wolnik 290229**

**Obraz zawierający zrzut ekranu, zieleń, wzór, Wielobarwność

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna. Obraz zawierający zrzut ekranu, zieleń, wzór, Wielobarwność

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.**

**1. Interpretacja zadania**

Zadanie polegało na stworzeniu programu symulującego rozprzestrzenianie się pożaru w lesie o różnej gęstości zadrzewienia. Kluczowe założenia:

* Las modelowany jako dwuwymiarowa siatka komórek
* Każda komórka może być: pusta, drzewem, płonąca lub spalona
* Pożar inicjowany jest przez "uderzenie pioruna" w losowe drzewo
* Ogień rozprzestrzenia się do sąsiednich komórek wg wybranej strategii (Moore'a lub von Neumanna)
* Program oblicza stabilną średnią procentową spalonych drzew dla danej gęstości

**2. Struktura projektu**

* **main.rs**: Punkt wejścia, parsowanie argumentów, zarządzanie symulacjami
* **config.rs**: Konfiguracja parametrów symulacji (rozmiar, gęstość, strategia spalania)
* **forest.rs**: Implementacja lasu (siatka, stany komórek, logika spalania)
* **fire\_spread.rs**: Strategie rozprzestrzeniania ognia (Moore i von Neumann)
* **simulation.rs**: Logika uruchamiania symulacji i zbierania wyników
* **display.rs**: Wizualizacja animacji w terminalu

**Kluczowe struktury danych:**

**enum** **CellState** { Empty, Tree, Burning, Burned }

**struct** **Forest** {

grid: Vec<Vec<CellState>>,

fire\_front: VecDeque<(usize, usize)>,

burned\_count: usize,

total\_trees: usize

}

**enum** **BurnPattern** {

**Moore**(MooreNeighborhood),

**VonNeumann**(VonNeumannNeighborhood)

}

**let** total\_cells = size \* size;

**let** target\_trees = (density \* total\_cells **as** f64).**round**() **as** usize;

**let** **mut** grid = vecvec![CellState::Empty; size]; size];

​​

**3. Reprezentacja lasu**

Struktura danych: Vec<Vec<CellState>>

Zalety:

* Czytelność indeksowania: forest.grid[x][y] od razu daje dostęp do stanu komórki w wierszu x i kolumnie y.
* Prosta implementacja i łatwość debugowania (w trybie „print” możemy wypisać każdą komórkę).

Wady:

* Rozmiar: size × size elementów wymaga bloku pamięci „dwuwymiarowego” (w Rust faktycznie alokowane jest Vec< Vec<…> >, czyli size oddzielnych wektorów po size elementów każdy). Przy 10000 × 10000 to 100 000 000 komórek, co zajmuje (zakładając 1 bajt na CellState, + kilka bajtów nadmiarowych) realnie około 200 MB pamięci.

Alternatywa: wektor liniowy *Vec<CellState>*

Można by przechowywać całą siatkę w pojedynczym wektorze o rozmiarze size \* size i konwertować współrzędne (x, y) na indeks idx = x \* size + y.

Pseudokod:

// Utwórz wektor wszystkich współrzędnych:

**let** **mut** indices: Vec<(usize, usize)> = Vec::**with\_capacity**(total\_cells);

**for** x **in** 0..size {

**for** y **in** 0..size {

indices.**push**((x, y));

}

}

// Mieszaj tablicę:

**let** **mut** rng = rand::**rng**();

indices.**shuffle**(&**mut** rng);

// Umieść pierwsze 'target\_trees' drzew:

**for** &(x, y) **in** indices.**iter**().**take**(target\_trees) {

grid[x][y] = CellState::Tree;

}

Zalety:

* Mniejszy overhead alokacji (jeden wektor zamiast *size* wektorów).
* Potencjalnie lepsza lokalność pamięci (jedna nieprzerwana struktura).

Wady:

* Konieczność każdorazowego obliczania indeksu (choć kosztem 1 × mnożenie + 1 × dodawanie, co jest bardzo szybkie).
* Trudniej czytelnie wypisać wierszami/kolumnami (przy rysowaniu animacji trzeba wtedy dodatkowo pętlować po *x* i *y*, wywołując index(x, y, size).

Decyzja:

Ze względu na czytelność kodu i łatwość debugowania wybrano *Vec<Vec<CellState>>*. W kolejnych iteracjach projektu, gdy zajdzie potrzeba obsługi większych rozmiarów   
(≥ 10000×10000), można będzie bez większego nakładu zmienić to na wektor liniowy.

**4. Inicjalizacja drzew – Fisher–Yates shuffle**

Dzięki zastosowaniu „Fisher–Yates” (algorytm optymalnego mieszania wektora) mamy pewność, że każda para *(x, y)* ma równe szanse pojawienia się wśród pierwszych *target\_trees* elementów. W rezultacie otrzymujemy dokładnie liczbę drzew równą *round(density \* size²)*, bez powtórzeń.

Pseudokod:

**let** total\_cells = size \* size;

**let** target\_trees = (density \* total\_cells **as** f64).**round**() **as** usize;

**let** **mut** grid = vec![vec![CellState::Empty; size]; size];

// Utwórz wektor wszystkich współrzędnych:

**let** **mut** indices: Vec<(usize, usize)> = Vec::**with\_capacity**(total\_cells);

**for** x **in** 0..size {

**for** y **in** 0..size {

indices.**push**((x, y));

}

}

// Mieszaj tablicę:

**let** **mut** rng = rand::**thread\_rng**();

indices.**shuffle**(&**mut** rng);

// Umieść pierwsze 'target\_trees' drzew:

**for** &(x, y) **in** indices.**iter**().**take**(target\_trees) {

grid[x][y] = CellState::Tree;

}

Uwagi:

Rozważono również inne metody (losowanie każdej komórki z prawdopodobieństwem *density*, jednak wtedy nie gwarantujemy dokładnej liczby drzew, co utrudnia porównywalność wyników między kolejnymi symulacjami.

**5. Propagacja ognia**

Zamiast przy każdym kroku iterować całą siatkę w poszukiwaniu pól `Burning`, utrzymujemy kolejkę *fire\_front: VecDeque<(usize, usize)>*, do której w miarę zapalania nowych drzew dopisujemy ich współrzędne. Dzięki temu w każdym kroku symulacji przetwarzamy wyłącznie pola będące faktycznie płonące (BFS – Breadth-First Search).  
Pseudokod:

repeat:

**if** fire\_front.**is\_empty**() then **break**

current\_burning = wszystkie elementy **fire\_front** (drain)

**for** **ka**ż**de** (x, y) **in** current\_burning:

oznacz grid[x][y] = Burned

burned\_count += 1

**for** **ka**żdy są**siad** (nx, ny) = strategy.**spread**(x, y, size):

**if** grid[nx][ny] == Tree:

grid[nx][ny] = Burning

fire\_front.**push\_back**((nx, ny))

until fire\_front.**is\_empty**()

Zalety:

* Gdy ogień rozprzestrzenia się powierzchniowo, kolejka przenosi tylko aktualnie płonące drzewa, a nie wszystkie drzewa z siatki.
* W pozostałych polach stan „Tree” pozostaje nietknięty, co minimalizuje liczbę niepotrzebnych porównań.

Złożoność czasowa:

* Budowa lasu (wypełnienie Vec<Vec<CellState>> i shuffle) ≈ O(size²).
* Symulacja rozprzestrzeniania ≈ O(burned\\_count + rozmiar frontu × liczba sąsiadów).   
  W najgorszym wypadku, gdy spłonie każdy las:  
  burned\_count = total\_trees = round(density \* size²).

**6. Struktura *FireSpreadStrategy***

Trait umożliwia w prosty sposób dodanie kolejnych sposobów rozprzestrzeniania ognia (np. heksagonalne pola lub modele z wiatrem).

W kodzie:

**pub** **trait** **FireSpreadStrategy** {

**fn** **spread**(&**self**, x: usize, y: usize, size: usize) -> Vec<(usize, usize)>;

}

Implementacje:

* MooreNeighborhood (8 kierunków)
* *VonNeumannNeighborhood* (4 kierunki)

Rozszerzalność:

Gdyby w przyszłości chciano dodać nowe strategie rozprzestrzeniania się ognia lub wprowadzić „wiatrowe” warunki, wystarczyłoby zaimplementować *spread*, które zwraca listę sąsiadów wraz z ewentualnymi wagami bądź prawdopodobieństwami.

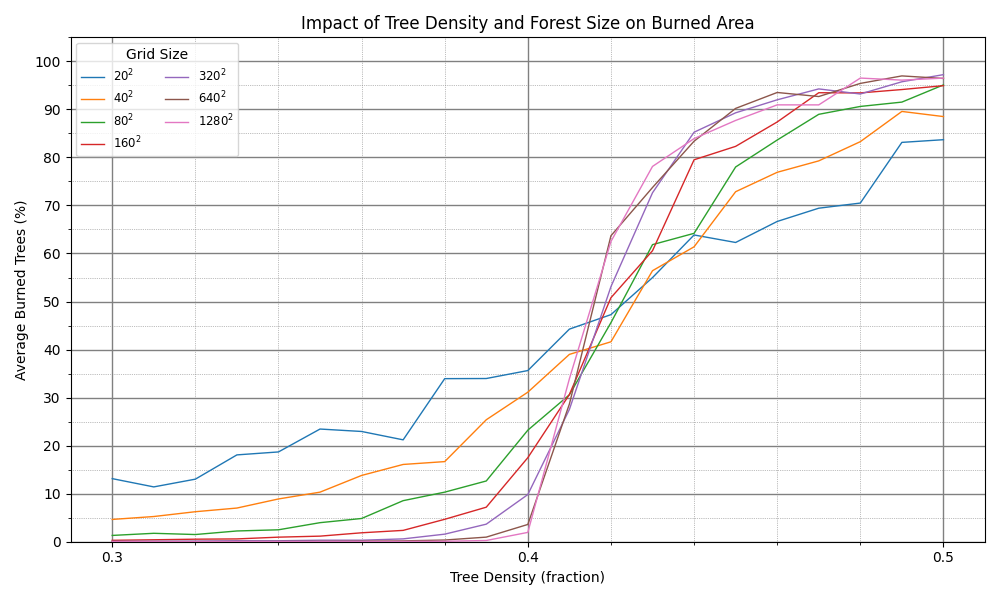
**7. Wyniki i analiza**

**Moore (8-kierunków)**:

* Powyżej 60% - spalenie prawie wszystkich drzew
* Największa zmiana: 35-55%
* Próg krytyczny: ~40%

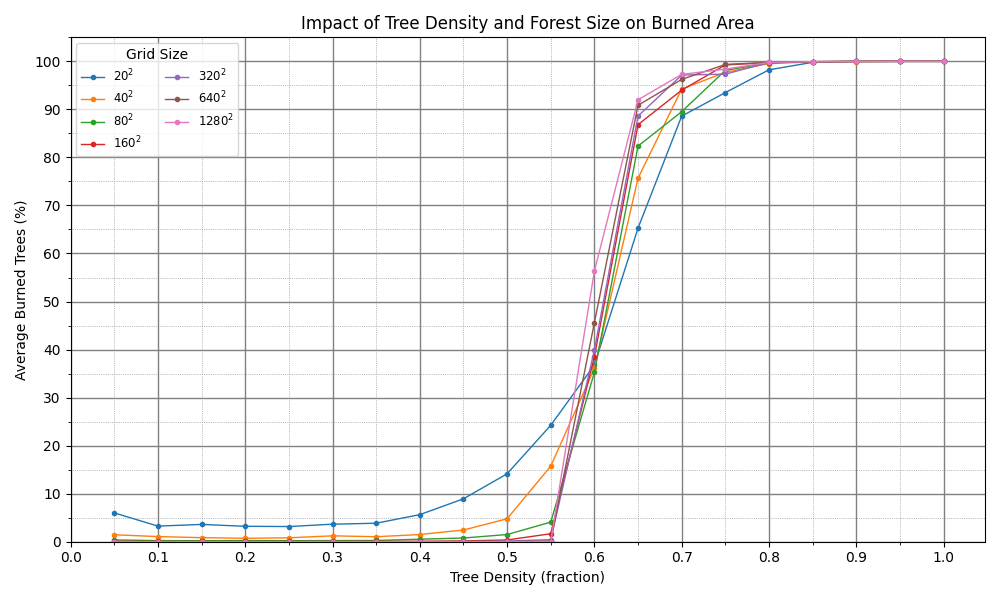
**![Obraz zawierający tekst, Wykres, linia, diagram

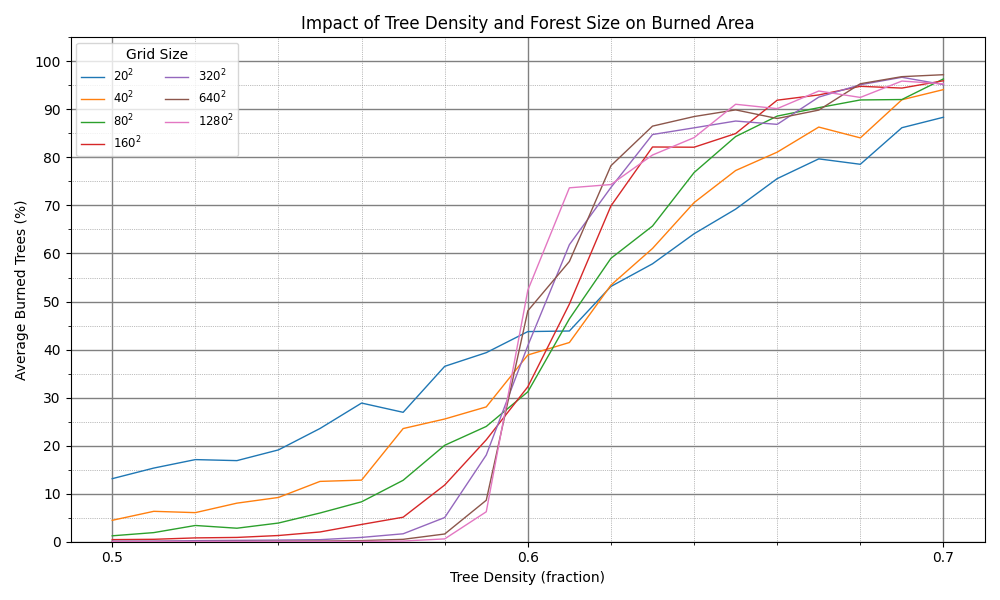
Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.**

****

**von Neumann (4-kierunki)**:

* Powyżej 80% - spalenie prawie wszystkich drzew
* Największa zmiana: 50-70%
* Próg krytyczny: ~55%



**Wnioski:**

* Im większy las tym drastyczniejszy przeskok w punkcie krytycznym
* Drastyczny podział gęstości lasów które średnie spalanie jest albo bardzo niskie albo bardzo wysokie

**8. Wizualizacja CLI**:

* Animacja z użyciem crossterm
* Symbole UTF-8 (🌲, 🔥, ◼️)

**9. Testowanie i walidacja**

* Testy jednostkowe dla wszystkich modułów
* Testy strategii sąsiedztwa (brzegi, rogi siatki)
* Walidacja argumentów CLI