12.06.2016
Maciej Wlazło
AiSD
Projekt zaliczeniowy: *Program generujący labirynty oraz znajdujący ścieżki wewnątrz wygenerowanych labiryntów.*

Opis:

Projekt został napisany w języku Java wykorzystując bibliotekę graficzną Swing aby wyświetlić graficznie efekty działania programu.
Labirynt to zbiór ścieżek, o skomplikowanym układzie, prowadzących od punktu początkowego do punktu końcowego. W moim generatorze labiryntów wykorzystuje sześcian o wymiarach NxN podzielony na N^2 pól. Pola te można uznać za węzły grafu i przechowywane są w strukturze tablicy węzłów oznaczonej jako array[][]. Węzły zaimplementowane są za pomocą klasy Cell, która rozszerza klasę JPanel. Każdy węzeł Cell posiada atrybuty:

- north, south, east, west – określa czy w danym kierunku znajduje się ściana
- visited – oznacza czy węzeł został już odwiedzony czy nie
- start – określa czy dany węzeł jest punktem początkowym(korzeniem)
- end – określa czy dany węzeł jest punktem końcowym
- border – określa czy danym węzeł jest częścią obramowania
- drawme – określa czy dany węzeł jest częścią ścieżki rozwiązania
- drawme2 – określa czy dany węzeł jest częścią ścieżki rozwiązania, którą program cofa się po już istniejącej ścieżce(backtracking).

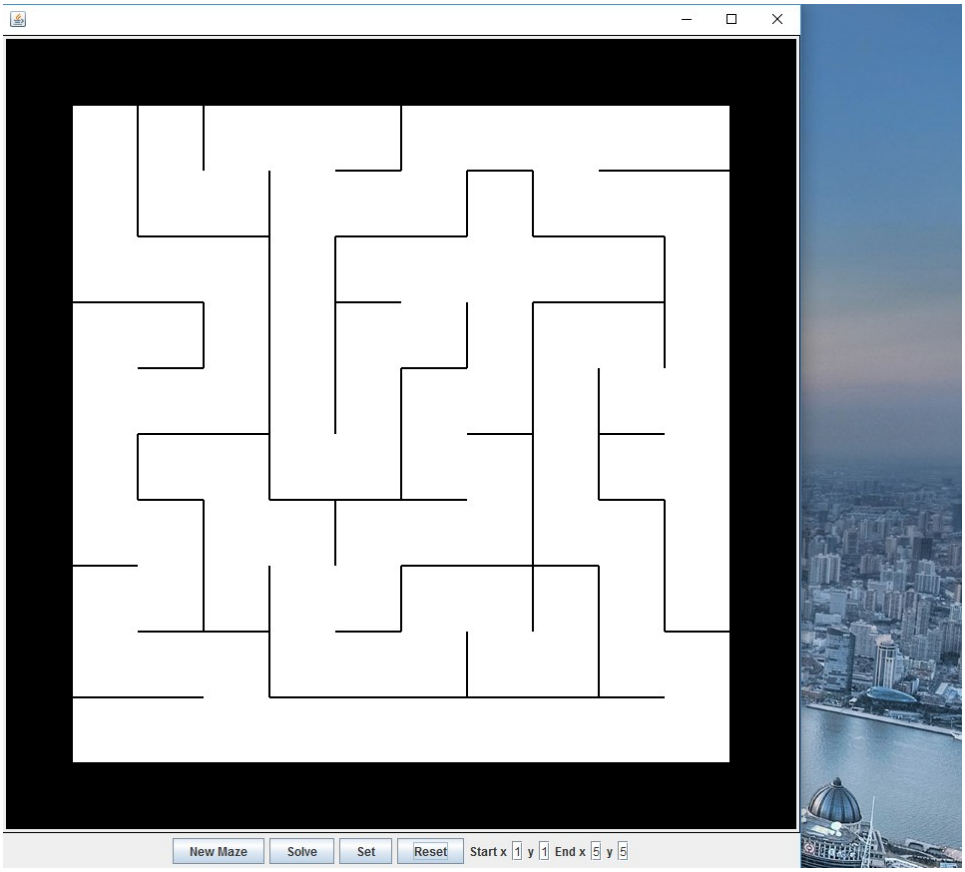
Dzięki temu każdy węzeł jest odpowiedzialny za przechowywanie informacji o swojej części labiryntu a nie zewnętrzna struktura.

Generowanie labiryntu:

Do generowania labiryntu najpierw korzystam z funkcji pregenerate(), która tworzy obramowanie wokół kwadratu NxN. Liczba N jest ustawiona na 10. Obramowanie zapobiega wychodzeniu indeksów poza dozwolony zakres oraz wyróżnia labirynt od reszty GUI aplikacji. Następnie wywoływana jest funkcja generate(). Funkcja ta działa podobnie do Depth-First-Search. Na początku we wszystkich polach tablicy array[][] każda ściana jest obecna i pole visited jest ustawione na false, analogicznie jak w algorytmie DFS na zajęciach, gdzie każdy węzeł miał na początku kolor WHITE. Następnie wybieramy pierwszy punkt i zaznaczamy jako visited. Jeśli istnieje nieodwiedzony sąsiad, to wybieramy liczbę losową za pomocą klasy Ugenerator. W zależności jaką liczbę wybraliśmy wywołujemy metodę generate() u odpowiedniego sąsiada, a wcześniej usuwamy ściany dzielące oba węzły. Zamiast wykorzystywać metodę Adjacent() wybieramy losowo sąsiada dla danego węzła. Dzięki temu na bieżąco określamy sąsiadów węzła i tworzymy labirynt. Dopóki program nie znajdzie się w ślepym zaułku(otoczonym z 3 stron polami gdzie visited=true) będzie tworzył jak najdalej daną ścieżkę. W przeciwnym wypadku zacznie się cofać rekurencyjnie i tworzyć alternatywne ścieżki tam gdzie to możliwe.

Zamiast rekurencji można również zaimplementować Stos.

Po pierwszym uruchomieniu programu aplikacja powinna wyglądać następująco:



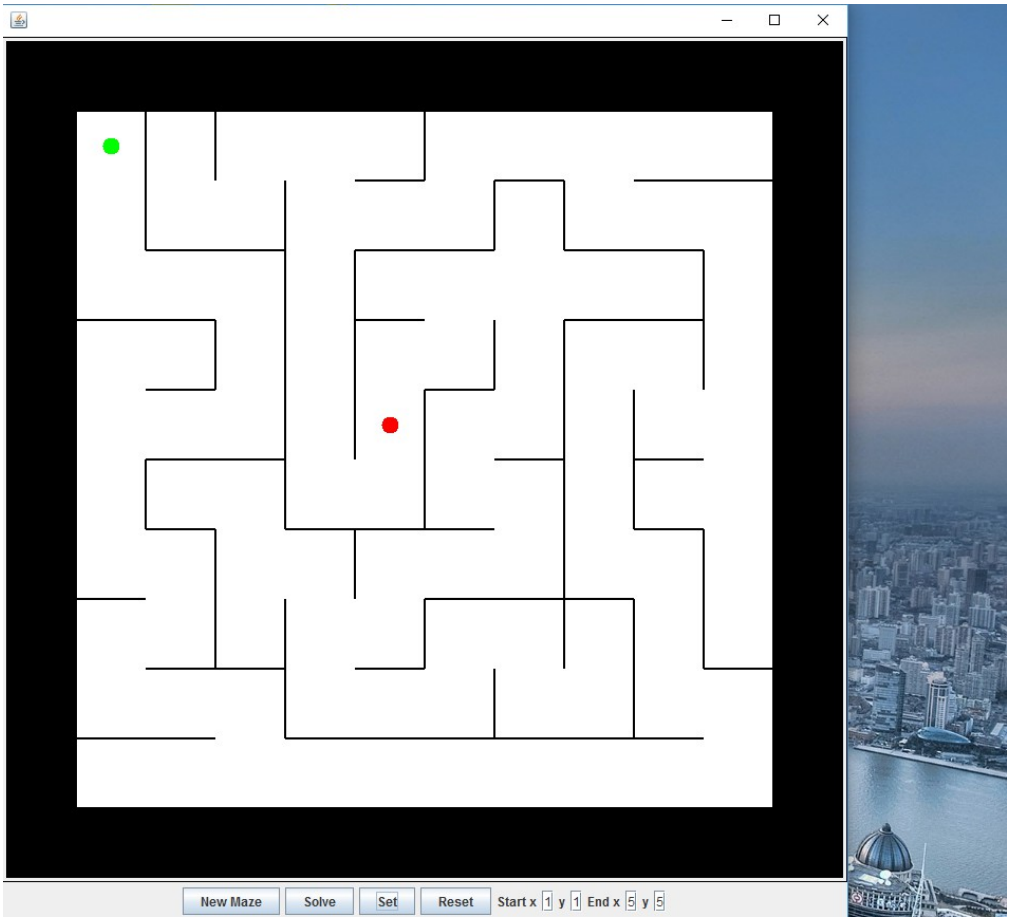
Ilustracja 1: Wygląd aplikacji po uruchomieniu

Ustawianie ścieżki:

Następnie można ustawić punkty początkowe oraz końcowe lub wykorzystać punkty domyślne(start(1,1) i end(5,5). Tablica numerowana jest w następujący sposób:

- Współrzędna x oznacza wysokość gdzie 1 to góra, a N to dół
- Współrzędna y oznacza szerokość gdzie 1 to lewa, a N to prawa strona
- Pierwsza para oznacza współrzędne punktu początkowego a druga para punktu końcowego

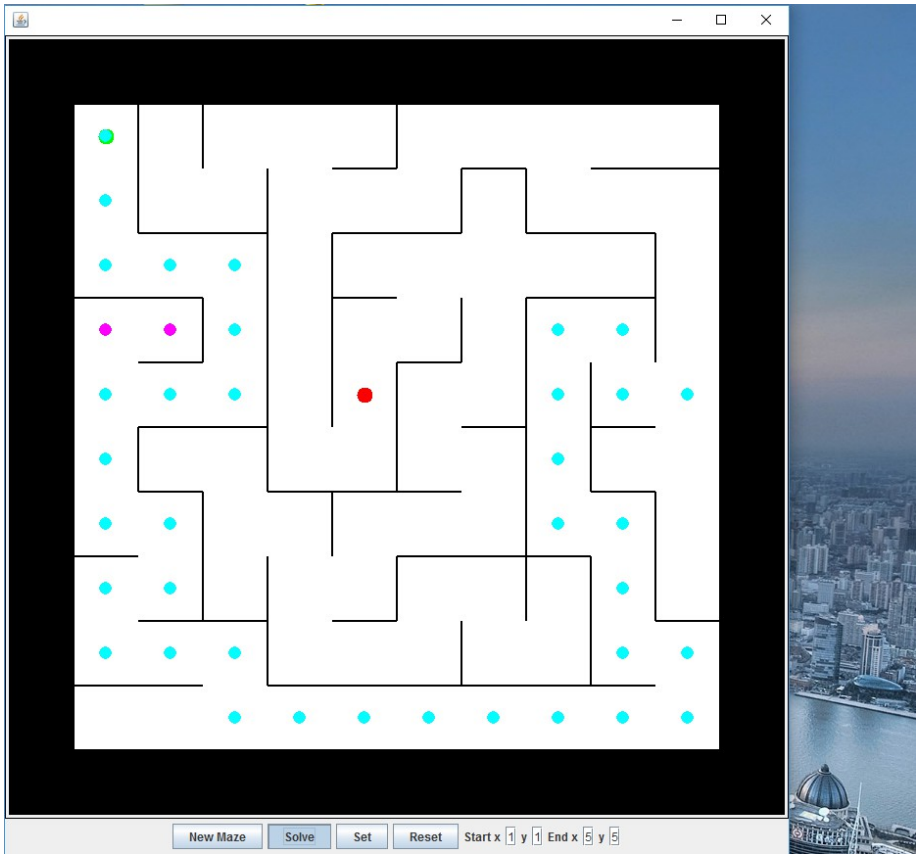
Po wybraniu przycisku Set labiryntu powinien zostać ustawiony.



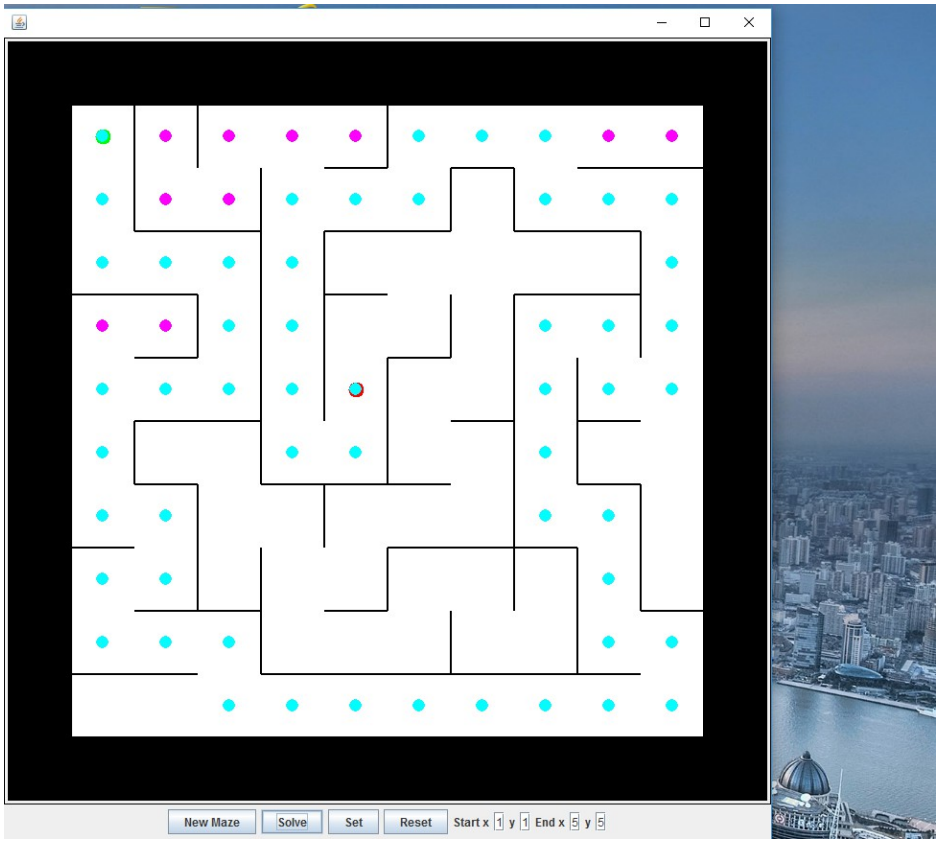
Ilustracja 2: Wygląd aplikacji po naciśnięciu SET

Rozwiązywanie labiryntu:

Do rozwiązywania labiryntu również wykorzystuję metodę podobną do DFS. Najpierw sprawdzane jest czy aktualne współrzędne nie są współrzędnymi obramowania. Ponieważ funkcja działa rekurencyjnie sprawdzane jest czy aktualny węzeł nie jest punktem końcowym lub czy nie był już odwiedzony. Po weryfikacji węzeł zostaje uznany za odwiedzony i rysowana zostaje kropka w pierwszym kolorze. Następnie następuje blok if-ów, który działa podobnie do metody Adjacent() i wybrany zostaje następny punkt poszukiwań. Jeśli w wyniku rekurencji zaczniemy się cofać po węzłach już zaznaczonych pierwszym kolorem, zostanie użyty drugi kolor oznaczający backtracking. Funkcja kontytuuje działanie dopóki nie znajdzie szukanego końcowego węzła.



Ilustracja 3: Aplikacja w trakcie wykonywania funkcji Solve



Ilustracja 4: Aplikacja po zakończeniu działania funkcji Solve

Resetowanie i generowanie nowego labiryntu:

Przyciski Reset oraz New Maze służą odpowiednio do resetowania i usuwania danych z ostatniego przebiegu funkcji solve(), oraz do tworzenia nowego labiryntu. Działają wykorzystując wyżej wymienione funkcje.