Sprawozdanie
Projekt Indywidualny

Daniel Daczko (299241)

30 listopada 2019

1. Część pierwsza

1.1. Struktura przechowująca labirynt

Za przechowywanie labiryntu odpowiedzialna jest klasa `Maze`. Znajduję się w niej tablica dwuwymiarowa, której zawartość jest odwzorowaniem pliku tekstowego, którego format jest następujący:

```
+#+++++++++
+00000+000+
+++++0+0+0+
+000+0+0+0+
+0+++0+++0+
+00000+000+
+++0+++0+++
+000+000+0+
+0+++0+0+0+
+00000+000+
+++++++++*+
```

gdzie: „`+`" to ściany, „`0`" to przejście, „`#`" to wejście i „`*`" to wyjście.

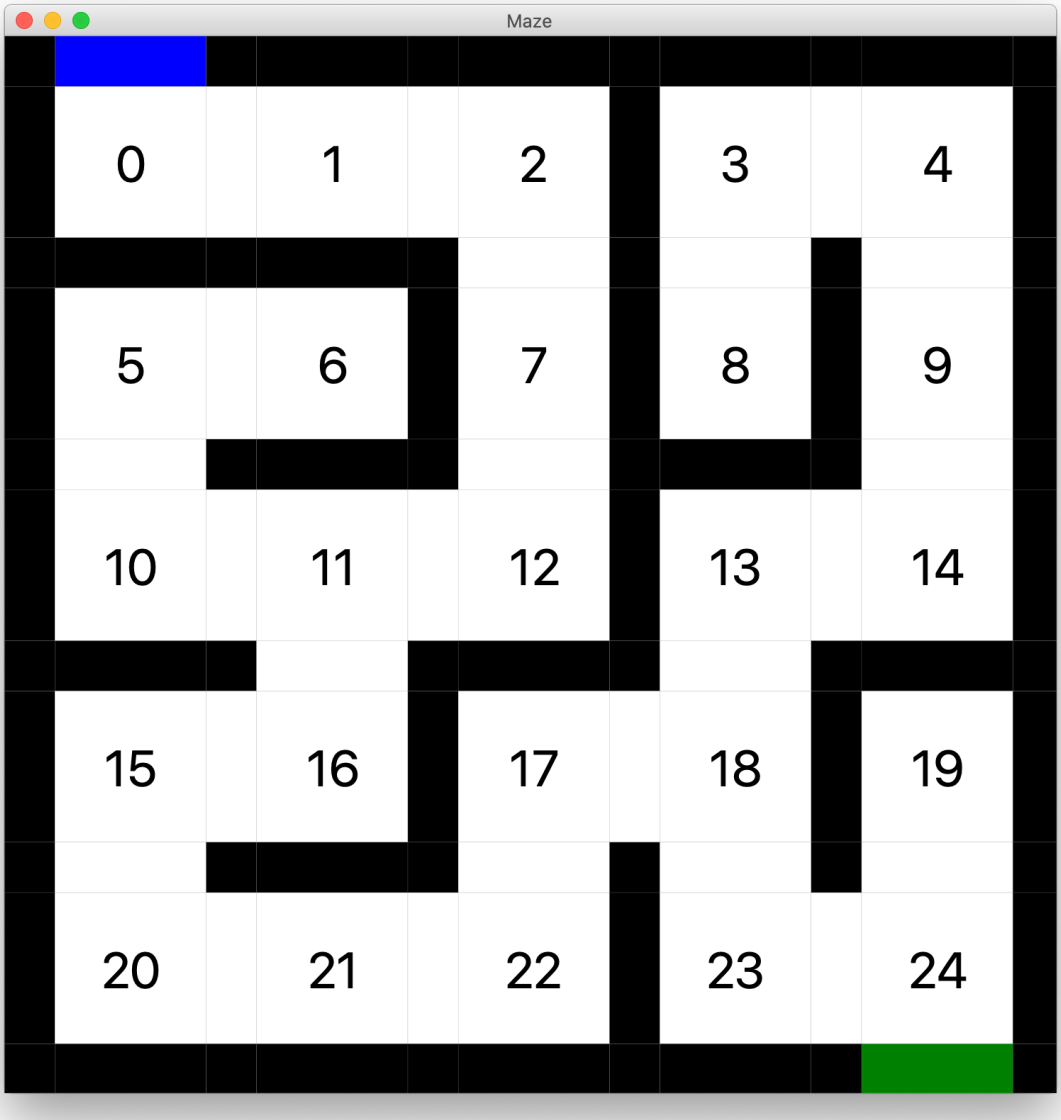
Wejście musi znajdować się w pierwszym, a wyjście w ostatnim wersie. Klasa ta posiada dwa konstruktory:

- pierwszy przyjmuje jako argumenty rzeczywistą wysokość i szerokość labiryntu (zakładając, że ściany są nieskończenie cienkie), następnie korzystając z algorytmu rekurencyjnego podziału tworzy losowo labirynt;
- drugi przyjmuje jako parametr ścieżkę do pliku tekstowego, który wykorzystuje metodę `readFromFile()` zapisując labirynt w postaci tablicy.

Klasa `Maze` posiada metodę `saveToFile()`, przy pomocy której możemy zapisać labirynt do pliku.

Labirynt może być również przechowywany w postaci grafu, co zostało zrealizowane w klasie `MazeAsGraph`, która w swym konstruktorze otrzymuje instancję klasy `Maze`. Tworzy ona listę węzłów (instancji klasy `Node`) oraz przechowuje bezpośredni dostęp do węzła

skojarzonego z wejściem i wyjściem z labiryntu. Węzły ponumerowane są w macierzy po kolei od lewej do prawej zaczynając od 0. Można to zaobserwować na poniższym rysunku.



Rysunek 1. Graficzna reprezentacja labiryntu

1.2. Algorytm rekurencyjnego podziału

Algorytm ten został zaimplementowany w klasie `RecursiveDivisionAlgorithm` w metodzie pod nazwą `division()`. Na samym wstępie metoda sprawdza czy wycinek labiryntu, który otrzymała jako argument jest jeszcze możliwy do podziału poprzez wstawienie ściany. Jeżeli jego rzeczywista szerokość lub wysokość jest mniejsza lub równa „1", metoda kończy działanie.

W przeciwnym wypadku tworzy w losowym dostępnym miejscu ścianę poziomą (gdy szerokość wycinka jest większa od wysokości) lub pionową (w przeciwnym wypadku.) Jeśli obie wielkości są sobie równe, losuje orientację.
W wylosowanym wcześniej miejscu wycina jedno przejście. Dodatkowo, by pokazać różnicę pomiędzy algorytmami wyszukiwania ścieżki wyjścia z labiryntu, metoda losuje i wycina drugie przejście z prawdopodobieństwem wystąpienia 33%.
Kolejnym krokiem jest podział w zależności od orientacji na części „lewą i prawą" lub „górną i dolną". Dla każdej z nich metoda wywołuje samą siebie z odpowiednimi parametrami.

2. Część druga

2.1. Algorytm Breadth First Search

Algorytm znajduje się w klasie `BreadthFirstSearhAlgorithm`, która posiada metodę statyczną `findTheExitPath()`. Metoda jako argument przyjmuje instancję klasy `MazeAsGraph` i zwraca listę przechowującą kolejne węzły, przez które należy przejść, aby dojść do wyjścia.

2.2. Algorytm Trémaux

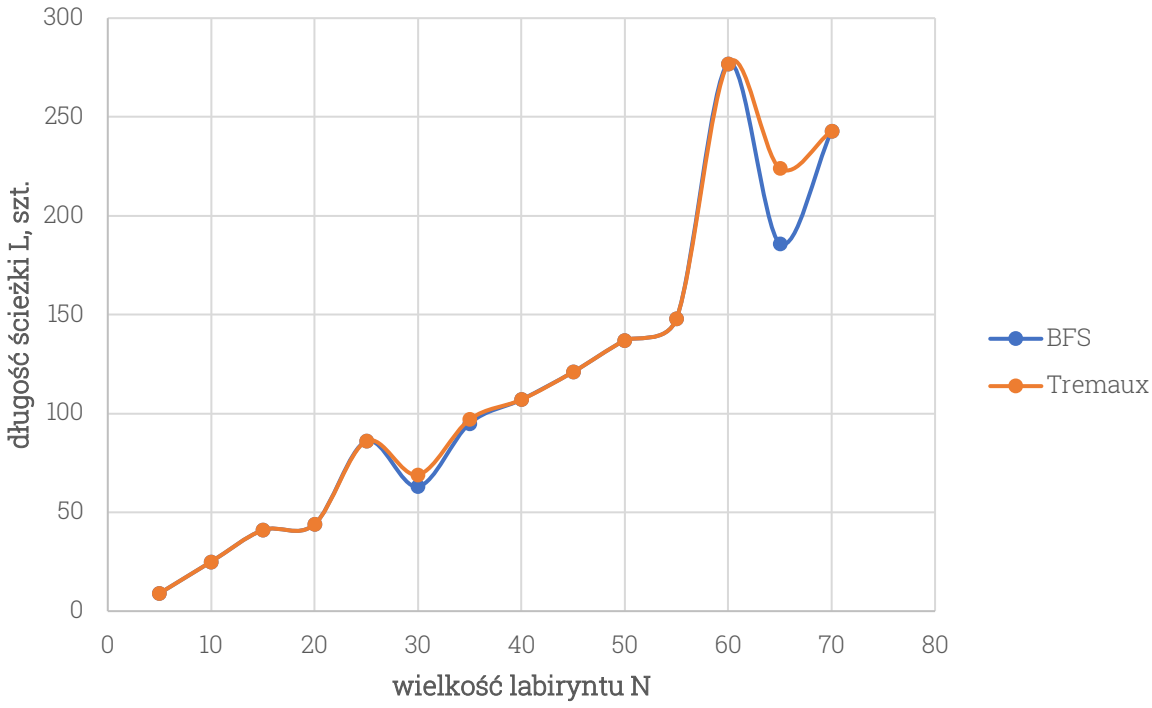
Algorytm znajduje się w klasie `TremauxAlgorithm`, która posiada metodę statyczną `findTheExitPath()`. Metoda jako argument przyjmuje instancję klasy `MazeAsGraph` i zwraca listę przechowującą kolejne węzły, przez które należy przejść, aby dojść do wyjścia.

2.3. Testy porównawcze

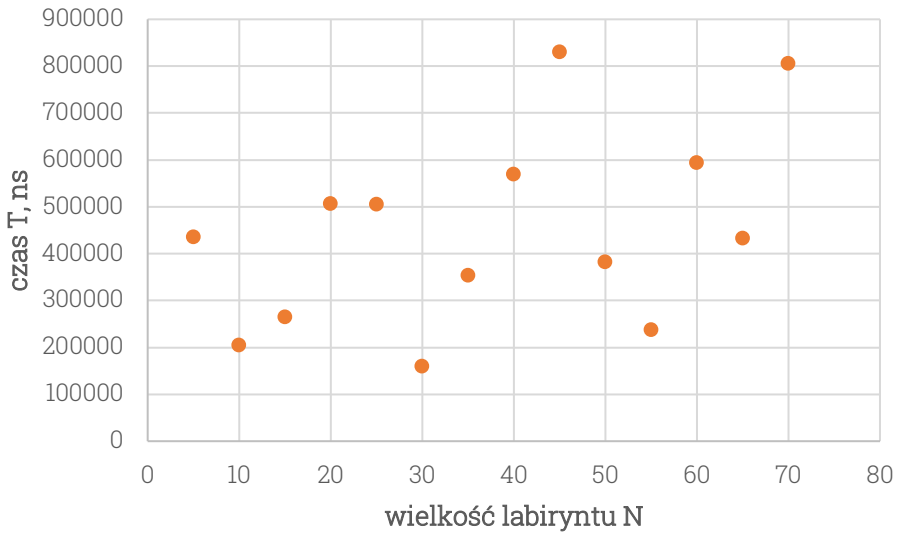
Do przeprowadzenia testów wykorzystałem 14 losowych labiryntów o wymiarach *„NxN"*. Dla każdego z nich sprawdziłem czas działania obu zaimplementowanych algorytmów *„T"* oraz długość *„L"* uzyskanej ścieżki. Wyniki można zobaczyć w *Tabeli 1.*
Na podstawie tej niewielkiej liczby danych można już zauważyć, że algorytm Trémaux jest znacznie szybszy od algorytmu BFS. Jednocześnie na *Rysunkach 3. i 4.* Można zaobserwować, że nie istnieje zależność pomiędzy wielkością labiryntu a czasem wyszukiwania ścieżki wyjścia, co potwierdza teorię.
Niestety ze względu na sposób swojego działania algorytm Trémaux, w przeciwieństwie do algorytmu Breadth First Search, nie zawsze znajduję najkrótszą ścieżkę.

Tabela 1. Wyniki pomiarów

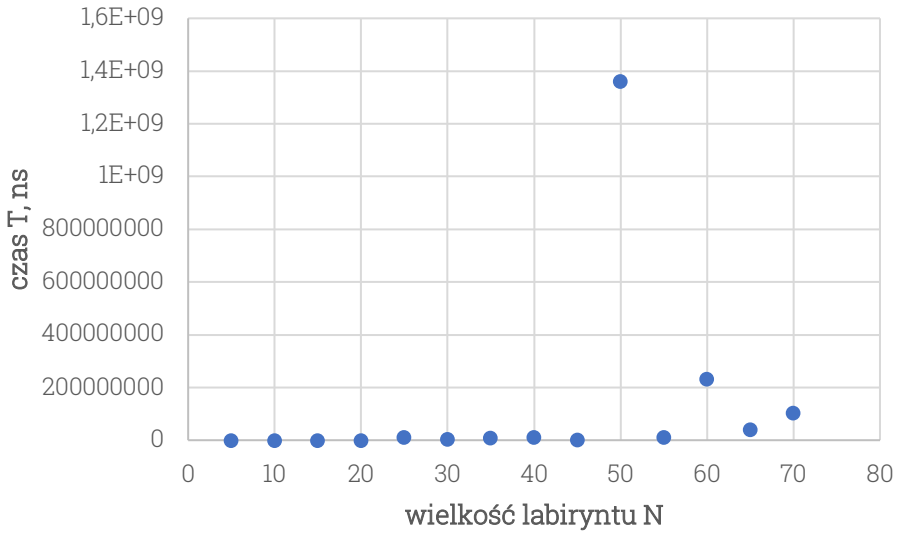
	Breadth First Search		Trémaux	
N	T, ns	L, szt.	T, ns	L, szt.
5	46464	9	436646	9
10	158546	25	204941	25
15	339708	41	265225	41
20	325275	44	506853	44
25	10755733	86	505515	86
30	4953186	63	161064	69
35	9339528	95	354117	97
40	10708890	107	569779	107
45	2236502	121	831298	121
50	1360411828	137	383186	137
55	11558215	148	238209	148
60	231753587	277	594744	277
65	41156504	186	432986	224
70	104519748	243	806527	243



Rysunek 2. Wykres zależności długości ścieżki wyjścia od wielkości labiryntu dla różnych algorytmów



Rysunek 3. *Wykres zależności czasu wyszukiwania ścieżki wyjścia od wielkości labiryntu dla algorytmu Trémaux*



Rysunek 4. *Wykres zależności czasu wyszukiwania ścieżki wyjścia od wielkości labiryntu dla algorytmu BFS*