Łukasz Laskowski nr. ind. 295803

Sprawozdanie końcowe

Projektu indywidualnego „Maze”

Spis treści:

1. Opis struktury projektu

- `FileManager` – jest to klasa odpowiadająca za czytanie plików oraz zapisywanie do nich gotowych labiryntów.
- `Maze` – klasa tworząca labirynt (na podstawie podanych z wejścia wymiarów lub ścieżki do pliku) i przechowująca go w dwuwymiarowej tablicy znaków.
- `BFSGraph` – w tej klasie tworzony jest graf na podstawie wyżej opisanej implementacji labiryntu, numerując wierzchołki, zaczynając od 0. Wypisywanie znalezionych ścieżek odbywa się jednak, dla czytelności, tak, jakby wierzchołki grafu numerowane były od 1.
- `TremauxGraph` – klasa ta robi kopię labiryntu, aby następnie móc znaleźć dowolną ścieżkę do wyjścia bezpośrednio w tablicy znaków.

2. Opis struktury labiryntu

a. Przechowywanie

Cały labirynt od początku przechowywany jest w dwuwymiarowej tablicy znaków. Wymiary tablicy to odpowiednio $2h+1$ i $2w+1$, gdzie h odpowiada wysokości, a w – szerokości labiryntu, podanej przez użytkownika. W tablicy mogą wystąpić 4 typy znaków – '0', '+', '*' i '#'. Oznaczają one kolejno przejścia, ściany, wyjście i wejście. Zakładamy, że wejście i wyjście znajdują się na przeciwległych sobie poziomych ścianach brzegowych.

b. Generowanie

Generacja odbywa się rekurencyjnie poprzez dzielenie aktualnej powierzchni roboczej labiryntu na dwie części, względem dłuższej ściany, a następnie wybranie jednego miejsca na tej ścianie, w którym ma się utworzyć przejście. Funkcja dzieląca wywołuje się następnie dla obu świeżo stworzonych pomieszczeń i robi to do osiągnięcia sytuacji, gdzie szerokość lub wysokość pomieszczenia będzie równa 1.

c. Budowa i numerowanie wierzchołków grafu

Graf w jawnej postaci zbioru wierzchołków i krawędzi występuje jedynie w algorytmie BFS. Z tego też względu jest on tworzony w klasie za BFS odpowiadającej.
W programie wykorzystałem listę sąsiedztwa. Tworzę tablicę list typu Integer, a w komórce o indeksie x znajduje się lista wszystkich jej sąsiadów. Listy te tworzę przechodząc po całej tablicy i sprawdzając, w którym kierunku istnieje przejście. Korzystam z zależności, że wierzchołki po lewej i po prawej stronie numery są odpowiednio mniejsze lub większe o 1 od aktualnie oglądanego, wierzchołek nad nim ma numer mniejszy o szerokość podaną przez użytkownika na wejściu, a pod nim – o tyle samo większy. W taki sposób otrzymuję listę sąsiedztw, którą następnie łatwo wykorzystać do przeszukania labiryntu algorytmem BFS.

3. Algorytmy szukania ścieżki w grafie – porównanie

- BFS – algorytm przeszukujący graf w szerz. Jeśli przyjmiemy konkretny wierzchołek grafu jako początek, a wyjście jako koniec, to możemy uznać, że robimy przejście po swojego rodzaju drzewie (pętli nie uwzględniamy, ponieważ BFS oznacza wierzchołki jako odwiedzone, gdy już je napotkał), którego korzeniem jest wejście, a wyjście jednym z liści. Następnie filtruję otrzymaną listę wierzchołków, idąc od jej końca. Otóż ostatnim na niej elementem jest wyjście z labiryntu, a więc wystarczy „obserwując" dany element listy usuwać te wierzchołki poprzedzające go w liście, które z nim nie sąsiadują w początkowym grafie i w taki sposób przesuwać się do początku listy. Po przejściu całej listy otrzymamy listę wierzchołków sąsiadujących ze sobą zarówno patrząc od początku, jak i od końca listy, a wiedząc, że BFS wpisywał je na listę w kolejności rosnącej odległości od korzenia, mamy pewność, że powstała lista wyznacza nam najkrótszą ścieżkę wejście-wyjście.

- Trémaux – algorytm ten operuje (w przypadku mojej implementacji) na tablicy znaków. Nie szuka on jednak najkrótszej ścieżki przejścia przez labirynt, natomiast próbuje po prostu jakoś się z niego wydostać. Odrzuca jednak możliwość zabłądzenia w labiryncie (oczywiście o ile w ogóle istnieje z niego wyjście) poprzez oznaczanie, z których przejść już skorzystał. W taki sposób finalnie otrzymujemy listę wierzchołków, po których zdarzyło nam się (nie jest to deterministyczny algorytm, ponieważ drogi są losowane spośród tych jeszcze godnych sprawdzenia) przejść po drodze do wyjścia, z pominięciem wszelkich ślepych uliczek, ponieważ po wejściu w takową wierzchołki usuwane są z listy wynikowej.

5. Sposób wyeksponowania różnic algorytmów przeszukujących graf

Wyżej opisane algorytmy w podstawowej wersji programu powinny dawać takie same wyniki. Różnica zacznie być zauważalna, gdy w labiryncie pojawią się kolejne możliwe trasy między wejściem a wyjściem. BFS nadal będzie zwracał najkrótszą z nich, podczas gdy Trémaux w pewnym sensie wylosuje sobie którąś z nich (możliwe jest też, że zwróci dokładnie tę samą ścieżkę). Sposobem na podkreślenie tej różnicy, który wykorzystałem w swoim programie było napisanie metody `makeRandomHoles(x: int)`, która wywołana w mainie po wygenerowaniu całego labiryntu utworzy x dodatkowych przejść, „dziurawiąc" ściany pionowe i poziome (na zmianę) i tworząc więcej możliwości wyjść. Funkcja ta daje gwarancję stworzenia dokładnie x dziur, w przeciwnym przypadku mogłaby nie trafić w żadną ścianę i pozostałaby wciąż tylko jedna ścieżka.