JIMP2 Sprawozdanie z projektu w języku Java

Stanisław Dutkiewicz, Filip Kobus

Maj 2024

Spis treści

1 Wstęp

Celem projektu jest opracowanie programu, który pozwala na znalezienie drogi we wczytanym labiryncie, na przykład najkrótszej ścieżki od wejścia do wyjścia. Labirynt jest udostępniany w postaci pliku tekstowego lub binarnego, a stworzony w Javie program, odpowiada za wczytanie labiryntu, znalezienie ścieżki oraz zapisanie labiryntu i ścieżki do pliku.

1.1 Problem Zadania

Głównym wyzwaniem projektu jest opracowanie przejrzystego interfejsu graficznego, który pozwoli użytkownikowi na bezproblemowe poruszanie się po funkcjonalnościach aplikacji oraz stworzenie efektywnej animacji przedstawiającej działanie algorytmów.

Dodatkowym wymaganiem jest efektywne zarządzanie zasobami systemowymi podczas przetwarzania dużych labiryntów, mimo że w przypadku aplikacji w języku Java nie ma bezpośredniego limitu zużycia pamięci. Ważne jest jednak, aby program działał sprawnie i szybko, nawet dla labiryntów o maksymalnym rozmiarze 1024x1024. Wyzwanie to wymaga implementacji skutecznych algorytmów przeszukiwania oraz optymalnego projektowania struktur danych używanych w programie.

1.2 Sposób rozwiązania

Do rozwiązania tego problemu wykorzystane zostaną dwie metody przeszukiwania: BFS (Breadth-First Search) oraz DFS (Depth-First Search), z których użytkownik może wybrać jeden. Algorytmy te posłużą do eksploracji labiryntu w celu znalezienia rozwiązania.

DFS jest to algorytm, który eksploruje labirynt, poruszając się zgodnie z ustalonymi wcześniej priorytetami ruchów (omijając przy tym ściany), a gdy napotka ślepy zaułek, cofa się do ostatnio napotkanego węzła, aby spróbować innej ścieżki. Algorytm został zaimplementowany w postaci iteracyjnej, mającej przewagę nad postacią rekurencyjną przez znacznie niższe zużycie zasobów.

BFS algorytm polega na przeszukiwaniu labiryntu wszerz, zapisując na stworzonej wcześniej kolejce możliwe do przejścia komórki, a następnie sukcesywnie odwiedzając każdą z nich co tym samym pozwoli na znalezienie najkrótsze drogi do wyjścia. Jego przewagą na algorytmem DFS jest pewność znalezienia najkrótszej ścieżki, co w DFS zależy od ustawień początkowych oraz konfiguracji ścian labiryntu.

Implementacja tych algorytmów w języku Java, którą opracowaliśmy, wymagała szczególnej uwagi na efektywne zarządzanie pamięcią. Aby sprostać temu wyzwaniu, kluczowe okazało się nie tylko zaprojektowanie samych algorytmów, ale również efektywne przechowywanie struktur danych, takich jak stosy i kolejki, które służą do zapamiętywania odwiedzonych komórek i ścieżek w labiryncie.

1.3 Rodzaje plików wejściowych

Plik tekstowy: Zawiera symbole reprezentujące elementy labiryntu tj. wejście do labiryntu (P), wyjście z labiryntu (K), ściany (X) oraz miejsca, po których można się poruszać (spacja).

Plik binarny: Składa się z nagłówka pliku, informującego o: rozmiarach labiryntu, współrzędnych wejścia i wyjścia oraz binarnych reprezentacjach elementów labiryntu (takich samych jak te w wersji tekstowej). Następnie zamieszczono w niej sam labirynt w postaci przypominającej wersję tekstową z niewielką różnicą tzn. przed informacją o rodzaju komórki (ściana lub wolna przestrzeń), umieszczona została liczba jego wystąpień w danym miejscu co pozwoliło na znaczne ograniczenie rozmiarów pliku wejściowego. Ostatnim elementem pliku jest sekcja rozwiązania gdzie autor labiryntu może zamieścić rozwiązanie do labiryntu, choć nie jest to wymagane.

2 Implementacja

Implementacja projektu, realizowanego w języku Java, skupiła się na rozwiązaniu problemu nawigacji przez labirynt z użyciem algorytmu przeszukiwania w głąb (DFS) oraz w szerz (BFS), przy jednoczesnym ograniczeniu zużycia pamięci. Kluczową decyzją projektową była modularna architektura systemu, która ułatwiła zarządzanie kodem oraz optymalizację pamięci. Poniżej są przedstawione kluczowe moduły projektu, które razem tworzą spójny system zdolny do skutecznego rozwiązywania labiryntów.

2.1 Moduły

Moduł `AlgorithmBfs`: Moduł `AlgorithmBfs` implementuje algorytm przeszukiwania wszerz (BFS) do znajdowania ścieżki w labiryncie.

- `queue`: kolejka przechowująca punkty do odwiedzenia.
- `runAlgorithm()`: metoda uruchamiająca algorytm BFS.
- `addPossibleMovesToQueue(Point point)`: metoda dodająca możliwe ruchy do kolejki.

Moduł `AlgorithmDfs`: Moduł `AlgorithmDfs` implementuje algorytm przeszukiwania w głąb (DFS) do znajdowania ścieżki w labiryncie.

- `stack`: stos przechowujący punkty do odwiedzenia.
- `makeMove()`: metoda wykonująca jeden krok algorytmu DFS.
- `getMove()`: metoda zwracająca ostatni wykonany ruch.

Moduł `Binary`: Moduł `Binary` odpowiada za konwersję labiryntu z formatu binarnego do formatu tekstowego.

- `readUnsignedShortLittleEndian(DataInputStream dis)`: metoda odczytująca 2 bajty z pliku binarnego w kolejności little-endian.

- `convertBinaryToText(String binaryFilePath, String textFilePath)`: metoda konwertująca plik binarny do formatu tekstowego.
- `generateMazeFromEncoding(DataInputStream binaryFile, BufferedWriter textFile, long wordCount, int separator, int wall, int path, int cols, int rows, int entryX, int entryY, int exitX, int exitY)`: metoda generująca labirynt na podstawie kodowania binarnego.

Moduł `DataArray`: Moduł `DataArray` reprezentuje strukturę danych przechowującą labirynt.

- `array`: tablica dwuwymiarowa przechowująca stan każdej komórki labiryntu.
- `putPointIntoArray(Point point)`: metoda umieszczająca punkt w tablicy.
- `setAsVisited(Point point)`: metoda oznaczająca punkt jako odwiedzony.

Moduł `Point`: Moduł `Point` reprezentuje punkt w labiryncie, przechowując jego współrzędne, typ i rodzica.

- `x`: współrzędna x punktu.
- `getX()`: metoda zwracająca współrzędną x punktu.
- `movePoint(int xDiff, int yDiff)`: metoda przesuwająca punkt o podane różnice.

Moduł `FileIO`: Moduł `FileIO` zajmuje się wczytywaniem i zapisywaniem labiryntów w formacie tekstowym i binarnym.

- `readMazeFromFile(File file)`: metoda wczytująca labirynt z pliku.
- `writeMazeToFile(BufferedImage image, File file)`: metoda zapisująca labirynt do pliku.
- `saveMazeAsText(BufferedImage mazeImage, JFrame window)`: metoda zapisująca labirynt jako plik tekstowy.

Moduł `MainGuiPanel`: Moduł `MainGuiPanel` jest odpowiedzialny za interfejs graficzny użytkownika oraz zarządzanie panelem głównym aplikacji.

- `mazeRenderer`: obiekt klasy `MazeRenderer` do renderowania labiryntu.
- `run()`: metoda uruchamiająca główny panel GUI.
- `openMaze()`: metoda otwierająca plik z labiryntem.

Moduł `MazeRenderer`: Moduł `MazeRenderer` odpowiada za renderowanie labiryntu w interfejsie graficznym.

- `mazeImage`: obraz labiryntu.

- `createMazePanel()`: metoda tworząca panel z labiryntem.
- `solveMazeWithBfs(DataArray dataArray)`: metoda rozwiązująca labirynt za pomocą algorytmu BFS.

Moduł `MazeUtilities`: Moduł `MazeUtilities` zawiera pomocnicze funkcje do obsługi labiryntu, takie jak sprawdzanie punktów wejścia i wyjścia.

- `setSelectedState(state: int)`: metoda ustawiająca stan wyboru punktu.
- `handleMazeCellSelection(MazeRenderer mazeRenderer, int imageX, int imageY, JFrame window)`: metoda obsługująca wybór komórki labiryntu.
- `resetEntranceAndExit(MazeRenderer mazeRenderer, JFrame window)`: metoda resetująca punkty wejścia i wyjścia w labiryncie.

Moduł `Main`: Moduł `Main` jest główną klasą uruchamiającą aplikację.

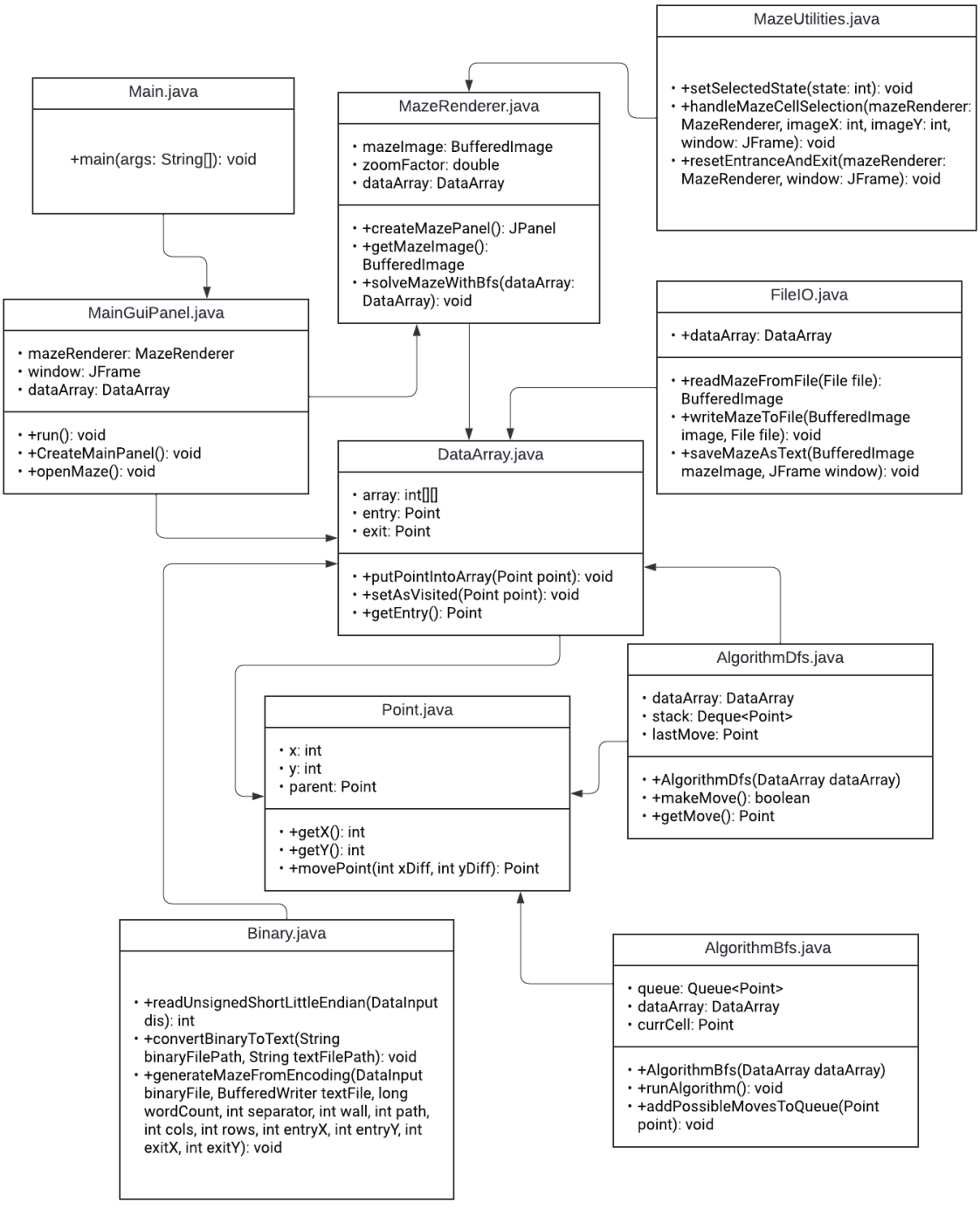
- `main(String[] args)`: metoda główna uruchamiająca aplikację.

2.2 Wzorce projektowe

- `Singleton`: Moduł `FileIO`: Klasa `FileIO` ma prywatny konstruktor, co sugeruje użycie wzorca Singleton, aby zapewnić, że tylko jedna instancja tej klasy będzie używana w programie.
- `Singleton`: Algorytmy `AlgorithmBfs` i `AlgorithmDfs`: Klasy te implementują różne algorytmy przeszukiwania (BFS i DFS). Można je traktować jako różne strategie rozwiązania problemu nawigacji w labiryncie, które można łatwo zamieniać w zależności od potrzeb.
- `Composite`: Klasa `DataArray` przechowuje strukturę labiryntu jako tablicę punktów (`Point`). Wzorzec Composite jest stosowany w sposobie, w jaki poszczególne komórki labiryntu są traktowane jako elementy składowe większej struktury.

2.3 Diagram Klas

Poniżej przedstawiono diagram klas stworzony w programie Lucidchart:



Rys.1 Diagram Klas projektu

3 Instrukcja użytkownika

Instrukcja użytkownika opisuje kroki, jakie należy wykonać, aby skorzystać z programu do rozwiązywania labiryntów w języku Java. Program ten pozwala na wczytanie labiryntu z pliku, wybranie algorytmu rozwiązania (BFS lub DFS), a następnie uruchomienie algorytmu w celu znalezienia ścieżki od wejścia do wyjścia. Poniżej przedstawiono szczegółowe instrukcje dotyczące korzystania z aplikacji.

3.1 Wymagania środowiskowe

Aby uruchomić program, należy mieć zainstalowane:

- Java Development Kit (JDK) w wersji 11 lub nowszej,
- System operacyjny: Windows, macOS lub Linux.

3.2 Instalacja

1. Pobierz pliki projektu z repozytorium lub innego źródła.
2. Rozpakuj pobrane archiwum do wybranego katalogu.
3. Otwórz terminal lub wiersz poleceń i przejdź do katalogu z projektem.

3.3 Uruchomienie programu

1. Skonfiguruj środowisko uruchomieniowe:
   - Upewnij się, że zmienne środowiskowe JAVA_HOME oraz PATH są poprawnie ustawione.
2. Otwórz katalog zawierający plik Main.java:

   ```
   cd src\main\java
   ```

3. Skompiluj kod programu, wpisując w terminalu:

   ```
   javac Main.java
   ```

4. Uruchom aplikację, wpisując w terminalu:

   ```
   java Main
   ```

3.4 Korzystanie z aplikacji

1. Po uruchomieniu programu, pojawi się główne okno aplikacji z menu oraz panelem do wyświetlania labiryntu.
2. Wybierz opcję **Plik -> Otwórz labirynt**, aby wczytać labirynt z pliku tekstowego lub binarnego.
3. Po wczytaniu labiryntu, możesz dostosować jego widok za pomocą przycisków do powiększania i pomniejszania.
4. Ustaw punkty wejścia i wyjścia, klikając na odpowiednie komórki labiryntu:
   - Wybierz opcję **Opcje -> Ustaw wejście i wyjście**.
   - Kliknij na wybraną komórkę na krawędzi labiryntu, aby ustawić punkt wejścia (oznaczony jako P).
   - Następnie kliknij na inną komórkę na krawędzi labiryntu, aby ustawić punkt wyjścia (oznaczony jako K).
5. Wybierz algorytm, którym chcesz rozwiązać labirynt:
   - **Opcje -> Znajdź najkrótszą ścieżkę (BFS)**, aby użyć algorytmu przeszukiwania wszerz.
   - **Opcje -> Wizualizuj szukanie ścieżki (DFS)**, aby użyć algorytmu przeszukiwania w głąb.
6. Po znalezieniu ścieżki, wynik zostanie wyświetlony na panelu z labiryntem.
7. Możesz zapisać wynikowy labirynt do pliku, wybierając odpowiednią opcję z menu **Plik**:
   - **Zapisz labirynt (txt)**, aby zapisać labirynt w formacie tekstowym.
   - **Zapisz obraz labiryntu**, aby zapisać labirynt jako obraz (PNG, JPEG, BMP).

3.5 Resetowanie i zatrzymywanie wizualizacji

- Aby zresetować ustawione ścieżki, wybierz **Opcje -> Resetuj ścieżki**.
- Aby przerwać trwającą wizualizację, również wybierz **Opcje -> Resetuj ścieżki**.

3.6 Zamykanie programu

Aby zamknąć program, kliknij przycisk zamknięcia okna.

4 Podsumowanie

Projekt, którego celem było opracowanie programu w języku Java, zdolnego do znajdowania ścieżek w labiryntach, został zrealizowany z sukcesem. Dzięki zastosowaniu algorytmów DFS oraz BFS, program jest w stanie skutecznie nawigować przez labirynty o różnych rozmiarach i złożonościach. Modularna architektura systemu oraz efektywne zarządzanie pamięcią pozwoliły na osiągnięcie wysokiej wydajności oraz skalowalności rozwiązania.

Podczas pracy nad projektem zdobyliśmy cenne doświadczenie w zakresie implementacji zaawansowanych algorytmów nawigacyjnych, zarządzania pamięcią w języku Java oraz testowania oprogramowania. Projekt ten stanowi solidną podstawę dla dalszych prac nad bardziej zaawansowanymi systemami nawigacyjnymi i optymalizacyjnymi.