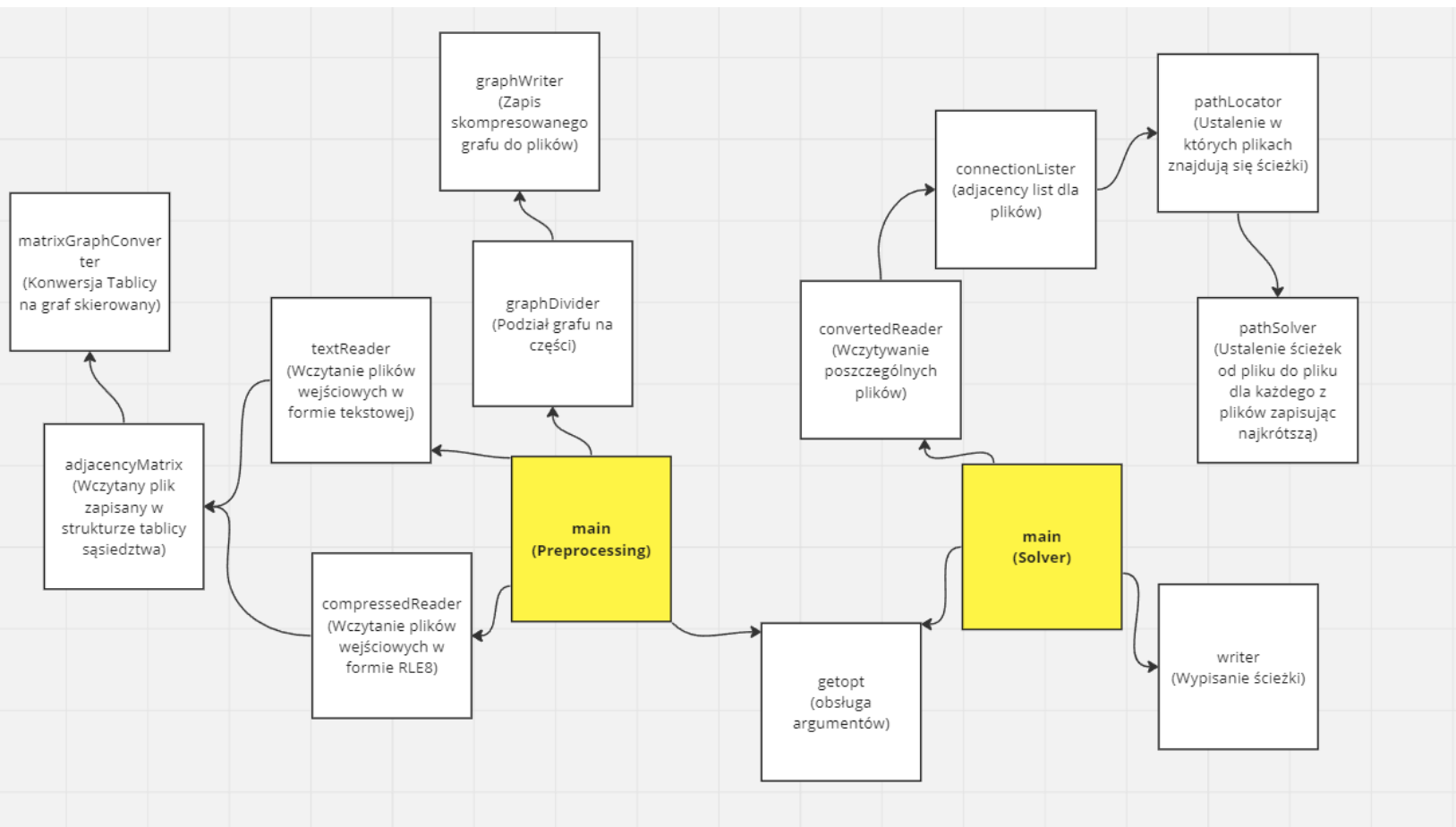


Specyfikacja implementacyjna

Damian Ciaszczyk, Karol Krukowski

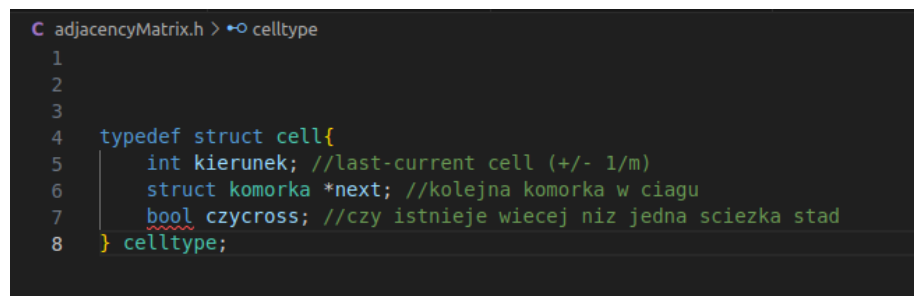
March 2024

1 Diagram modułów



2 Opis struktur danych

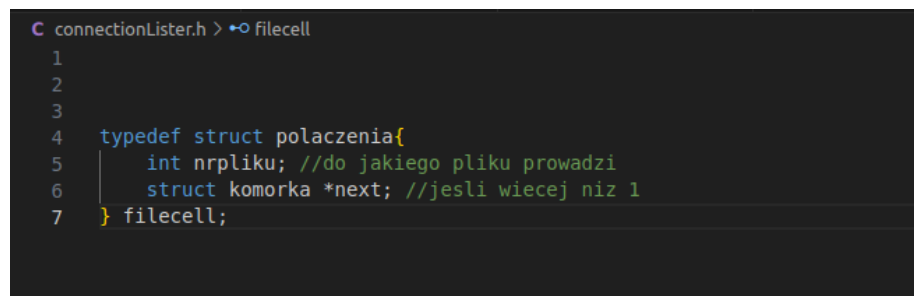
Występujące struktury:



```
C adjacencyMatrix.h > ➦ celltype
1
2
3
4 typedef struct cell{
5     int kierunek; //last-current cell (+/- 1/m)
6     struct komorka *next; //kolejna komorka w ciagu
7     bool czycross; //czy istnieje wiecej niz jedna sciezka stad
8 } celltype;
```

Rysunek 1: struktura wykorzystywana do listy sąsiedztwa oraz grafu

Ta struktura jest wykorzystywana najpierw jako tablica sąsiedztwa, w formie dwuwymiarowym gdzie każdy pojedyncza jednostka tej struktury odpowiada komórce labiryntu, w tym formacie znaczenie ma tylko lista *next przechowująca numery wszystkich komórek do których się łączy. Następnie, gdy jest wykorzystywana jako graf, opis jest zgodny z komentarzami na zdjęciu, gdzie informacja czy jest więcej niż jedna ścieżka pozwala na bezpieczne sprawdzenie next++.



```
C connectionLister.h > ➦ filecell
1
2
3
4 typedef struct polaczenia{
5     int nrpliku; //do jakiego pliku prowadzi
6     struct komorka *next; //jesli wiecej niz 1
7 } filecell;
```

Rysunek 2: Lista sąsiedztwa dla plików

Ta struktura to prosta lista przechowująca dla poszczególnego numeru pliku listę numerów plików do których prowadzi.

3 Algorytm

Algorytm który wykorzystamy do rozwiązania labiryntu to DFS (depth-first search). Jak sama nazwa wskazuje, opiera się on na przeszukiwaniu grafu najpierw w głąb, co biorąc pod uwagę "odchudzoną" listę plików która jest w naszym programie poddawana tego algorytmowi nie powinno zbyt obciążać systemu pamięciowo ani czasowo. Zdecydowaliśmy się na niego ponieważ nie wymaga

przechowywania dużej ilości danych, pasując idealnie do ograniczeń pamięci związanych z tym projektem. Oczywiście nie jest on idealnie dostosowany do labiryntów o wielu ścieżkach. Z tego względu, ponieważ można założyć że inne ścieżki muszą gdzieś wychodzić i wchodzić w "główną" ścieżkę labiryntu, jedyne co musimy wziąć pod uwagę aby znaleźć najkrótszą to wybrać która z tych alternatywnych dróg jest krótsza. To można zrobić uruchamiając równoległe mniejsze procesy tego samego algorytmu sprawdzające długość ścieżki jednocześnie ją rozwiązując. O ile byłoby to wtedy rozwiązanie rekurencyjne co nie sprzyja pamięci, to zakładając że nie jest to jakiś labirynt bez żadnych wewnętrznych ścian powinno to sprawdzać się doskonale. W wypadku takiego labiryntu może być jednak wymagane zaimplementowanie jakiegoś rodzaju kolejki.

4 Funkcje w poszczególnych modułach

Dla preprocesora:

1. Moduły `compressedReader` oraz `textReader`:
 - zawierają funkcjonalnie identyczne funkcje odpowiednio `CompRead()` i `StdRead()` przyjmujące jako argument nazwę pliku oraz strukturę do przechowania grafu, i konwertujące ten plik na tablicę sąsiedztwa dla dalszej obsługi. Jedyna różnica między nimi to to w jakim formacie obsługują labirynt.
2. Następnie w module `matrixGraphConverter`:
 - znajduje się funkcja `conv2graph()` przyjmująca za argumenty strukturę labiryntu oraz tą na graf, i tak jak nazwa wskazuje przekształca tablicę sąsiedztwa na graf.
3. W module `graphDivider`:
 - znajduje się funkcja `divide()` przyjmująca za argument strukturę grafu oraz nazwę pliku wynikowego. ta funkcja dzieli graf na mniejsze fragmenty, które są następnie zapisywane.
4. Za zapisywanie tych fragmentów odpowiada moduł `graphWriter`:
 - wykorzystujący do tego funkcję `PartWrite()`, przyjmującą za argumenty numer pliku, gałąź która będzie zapisywana oraz przedrostek pliku wynikowego.

Dla Solvera:

1. W module `convertedReader`:
 - `convRead()` przyjmująca za argument przedrostek plików trzymających dane o grafie, oraz wymiary labiryntu i liczbę plików, tworzy listę połączeń między poszczególnymi plikami, tworząc uproszczony labirynt.
2. Następnie moduł `pathLocator`:
 - Funkcją `LocatePath()` przyjmującą za argumenty przedrostek plików oraz listę ich zależności określa które pliki są potrzebne do znalezienia ścieżek w

labiryncie, następnie przekazując tak zmodyfikowaną listę do następnego modułu.

3. Moduł pathSolver:
 - wykorzystuje funkcję finalSolve() przyjmującą za argument po raz kolejny zmodyfikowaną listę zależności oraz przedrostek plików, wykorzystując algorytm dfs rozwiązuje poszczególne pliki tworząc ostateczne rozwiązanie.
4. Moduł Writer:
 - wykorzystujący zmienną writePath() przyjmującą nazwę pliku wynikowego, wypisuje w liście kroków jak przejść labirynt..

Funkcje wspólne w modułach getopt oraz Errormsg:

1. W getopt:
 - parsearg() przyjmująca liczbę argumentów wywołania, argumenty oraz listę flag do obsługi. W zależności od argumentów wywołania albo przekazuje odpowiednie informacje pokroju nazw plików dalej do programu albo zwraca informację o potrzebie wyświetlenia readme programu.
2. W Errormsg:
 - Returnmessage() przyjmująca jedynie kod błędu do wyświetlenia. Wyświetla błąd i albo pozwala programowi działać dalej albo wymusza zatrzymanie (w zależności od powagi błędu).