PSZT Projekt 1 – MM.P4 Labirynt

1. Treść zadania

Napisać program porównujący działanie algorytmów przeszukiwania BFS, DFS, IDFS dla problemu znalezienia drogi w labiryncie. Przestrzeń dyskretna - dozwolone ruchy to góra, dół, lewo, prawo. WE: plik ze strukturą labiryntu z we/wy (dla większych użyć jakiegoś generatora). WY: znaleziona najkrótsza ścieżka i/lub mapka z zaznaczoną ścieżką.

1. Przyjęte założenia
   * Labirynt może mieć dowolny rozmiar
   * Wygenerowany labirynt nie zawiera cykli (ścieżki prowadzącej z powrotem do ścieżki na której już byliśmy)
2. Podział odpowiedzialności
   * Paweł Świątkowski
     + Wczytywanie z pliku struktury labiryntu
     + Implementacja BFS
     + Implementacja prostego interfejsu użytkownika (menu)
     + Implementacja porównywania algorytmów
   * Grzegorz Aleksiuk
     + Zapisywanie do pliku struktury labiryntu
     + Implementacja DFS oraz IDFS (wspólna metoda o zadanej głębokości przeszukiwania)
     + Implementacja generatora struktury labiryntu
     + Implementacja wypisywania struktury i rozwiązania labiryntu
   * Wspólna praca
     + Ustalenie funkcjonalności
     + Ustalenie formatu danych (struktury w której przechowujemy labirynt)
     + Ustalenie interfejsów
     + Analiza algorytmów BFS, DFS, IDFS
3. Opis algorytmu
   * BFS
     + Tworzymy listę Visited zawierającą informację który węzeł został odwiedzony oraz kolejkę węzłów do odwiedzenia
     + Wrzucamy na kolejkę węzeł będący wejściem do labiryntu
     + Bierzemy z kolejki pierwszy element i zaznaczamy do w liście Visited jako odwiedzony
     + Jeżeli aktualny węzeł jest naszym celem (wyjściem z labiryntu) zakończ algorytm
     + Wrzucamy na kolejkę wszystkie węzły do których możemy dojść z aktualnego węzła, pod warunkiem że nie mają zaznaczone w liście Visited statusu odwiedzony
     + Skocz do pkt 3
   * DFS
     + Tworzymy listę Visited zawierającą informację który węzeł został odwiedzony oraz stos węzłów odwiedzonych (używam algorytmu iteracyjnego ze względu na „płytki” stos wywołań rekurencyjnych w JVM)
     + Wrzucam na stos węzeł będący wejściem do labiryntu
     + Bierzemy ze stosu element i zaznaczamy w liście Visited jako odwiedzony
     + Jeżeli aktualny węzeł jest naszym celem ( wyjściem z labiryntu) zakończ algorytm
     + Wrzucamy na stos pierwszy element do którego możemy dojść z aktualnego węzła, pod warunkiem że nie są zaznaczone w liście Visited jako odwiedzone. Jeżeli nie możemy wrzucić jakiegokolwiek elementu to zdejmij węzeł ze stosu
     + Skocz do pkt 3
   * IDFS
     + Dla kolejnych wartości depth (od 1 do x\*y) powtarzaj:
     + Tworzymy listę Visited zawierającą informację który węzeł został odwiedzony oraz stos węzłów odwiedzonych
     + Wrzucam na stos węzeł będący wejściem do labiryntu
     + Bierzemy ze stosu element i zaznaczamy w liście Visited jako odwiedzony
     + Jeżeli aktualny węzeł jest naszym celem ( wyjściem z labiryntu) zakończ algorytm
     + Jeżeli aktualny rozmiar stosu jest równy depth zdejmij element i skocz do pkt 4.
     + Wrzucamy na stos pierwszy element do którego możemy dojść z aktualnego węzła, pod warunkiem że nie są zaznaczone w liście Visited jako odwiedzone. Jeżeli nie możemy wrzucić jakiegokolwiek elementu to zdejmij węzeł ze stosu
     + Skocz do pkt 4
4. Raport
5. Wnioski

Początkowo sądziliśmy że BFS będzie osiągał najkrótsze czasy. Z eksperymentów wynika jednak, że to DFS średnio osiąga najlepsze wyniki. BFS odwiedza zwykle mniejszą liczbę wierzchołków niż DFS, jednak ze względu na konieczność uzupełniania listy backtrackingu w BFS (potrzebne aby znaleźć poszczególne węzły ścieżki) jest znacznie dłuższy czas przetwarzania pojedynczego wierzchołka w BFS niż w DFS, skutkuje to przewagą czasową DFSa pomimo większej liczby odwiedzonych wierzchołków. Złożoność pamięciowa jednego jak i drugiego jest porównywalna, ponieważ zakładamy możliwość wystąpienia cykli co za tym idzie dla obydwu algorytmów wymagana jest lista Visited (lista odwiedzonych wierzchołków).

Właściwością DFSa jest to, że działa skutecznie w ograniczonych grafach (nie radzi sobie w grafach nieskończonych). DFS ma znacznie lepszą złożoność pamięciową w drzewach (w nich nie ma cykli). BFS średnio sobie radzi w drzewach szerokich, jednak w naszym przypadku każdy wierzchołek ma co najwyżej 3 dzieci, więc BFS nie uzyskuje takich strat względem DFS.

W naszej implementacji IDFS ma podobną złożoność pamięciową jak BFS i DFS. Jednak uzyskuje znacznie wolniejsze wyniki niż pozostałe dwa algorytmy, ze względu na wielokrotne przechodzenie tych samych węzłów oraz wielokrotną inicjalizację tablicy odwiedzonych.

Rekurencyjna implementacja IDFSa rzeczywiście ma mniejszą złożoność pamięciową niż BFS.

1. Instrukcja użytkownika
   * Aby uruchomić program należy go zbudować ‘mvn clean install’ następnie przejść do folderu target i wywołać komendę ‘java -jar MazeSolver-1.0.jar’
   * Poruszanie się po menu polega na wpisywaniu odpowiednich liczb odpowiadających numerom poleceń
   * Dostępne polecenia:
     + „1 Read maze structure from file” – po wpisaniu nazwy pliku (bez rozszerzenia .txt) zawierającego strukturę labiryntu, jest ona ładowana do programu
     + „2 Generate maze structure” – generuje losowy układ labiryntu o zadanym rozmiarze przez użytkownika
     + „3 Print stats” – po wpisaniu rozmiaru labiryntu i ilości powtórzeń, program testuje algorytmy DFS, BFS, IDFS i wypisuje średni czas wyszukania rozwiązania i całkowitą ilość kroków
     + „0 Exit” – zakończenie pracy programu
   * Po wybraniu opcji 1 lub 2 pojawia się nam menu drugiego poziomu
     + „1 Print maze structure” – rysuje układ labiryntu w konsoli
     + „2 Show maze solution” – pokazuje rozwiązanie labiryntu w konsoli
     + „3 Print stats” - wypisanie czasu wykonania i liczby kroków dla algorytmów BFS, DFS i IDFS dla aktualnego labiryntu
     + „4 Save structure to file” – po wpisaniu nazwy pliku przez użytkownika, zapisywana jest aktualna struktura w formacie, który można następnie zaimportować do tego programu
     + „0 Main manu” – powrót do głównego menu