Tomasz Jankowiak 249006 17.05.2020

**Projektowanie Algorytmów i Metody Sztucznej Inteligencji**

Projekt 2 – Grafy

|  |  |
| --- | --- |
| Prowadzący: | mgr inż. Marta Emirsajłow |
| Termin zajęć: | Piątek, 9.15 |
| Numer grupy projektowej: | E08-06i |

1. **Wprowadzenie**

**Teoria grafów** ma zastosowanie w wielu dziedzinach życia, zadaniach praktycznych i zagadnieniach teoretycznych. Oprogramowanie oparte na analizie grafów znalazło zastosowanie między innymi w wyznaczaniu trasy pomiędzy punktami na mapie, czy najszybszej drogi ewakuacji z kompleksu budynków, a przedstawienie sieci komputerowych w postaci grafów pozwoliło na stworzenie programów usprawniających przepływ pakietów w Internecie. Teoria grafów jest jedną z najpotrzebniejszych dziedzin matematyki w informatyce.

Za pierwszą pracę na temat teorii grafów uznaje się referat naukowy na temat Zagadnienia Siedmiu Mostów napisany w 1736 przez szwajcarskiego matematyka Leonharda Eulera, który rozwiązał problem za pomocą grafów.

**Graf** jest strukturą danych składającą się z dwóch zbiorów: zbioru wierzchołków (węzłów) i zbioru krawędzi, które je łączą. Grafy mogą być reprezentowane na wiele sposobów. Najbardziej naturalnym i najprostszym dla człowieka jest rysunek grafu, jednakże dla komputera potrzebne jest coś innego. Dwa najpopularniejsze sposoby reprezentowania grafu w pamięci komputera to macierz sąsiedztwa oraz listy sąsiedztwa.

**Macierz sąsiedztwa** to macierz kwadratowa o stopniu n, gdzie n oznacza liczbę wierzchołków w grafie, reprezentowana za pomocą tablicy dwuwymiarowej. Wiersze oznaczają węzły początkowe krawędzi grafu, a kolumny ich węzły końcowe. Dla grafu nieskierowanego macierz sąsiedztwa jest symetryczna względem głównej przekątnej. W komórkach tablicy zapisane są wagi poszczególnych krawędzi grafu. Jeśli krawędź nie istnieje, można przypisać wartość 0 albo NULL. Główną zaletą reprezentacji tablicowej jest prostota implementacji programowej. Ponadto bezpośredni i tani (O(1)) dostęp do informacji: aby sprawdzić, czy dwa węzły są połączone krawędzią, wystarczy odczytać odpowiednią komórkę tablicy (stały czas dostępu, niezależny od położenia węzła ani rozmiaru grafu). Wadą jest to, że jedynie grafy o ustalonej z góry liczbie węzłów mogą być łatwo reprezentowane w postaci tablic (kłopotliwe i nieefektywne jest dodawanie i usuwanie węzłów). Ponadto, gdy nasz graf posiada małą liczbę krawędzi, to stopień efektywnego wykorzystania tablicy jest bardzo niski i większość komórek pozostanie pusta. Wymagania pamięciowe: **O(V2)**. W macierzach sąsiedztwa w porównaniu do list łatwo sprawdzić występowanie danej krawędzi, ale rozmiar użytej pamięci jest asymptotycznie większy.

**Lista sąsiedztwa** często okazuje się najefektywniejszą reprezentacją grafu. Dla każdego wierzchołka zapamiętywana jest lista sąsiadujących z nim węzłów. Wykorzystujemy tablicę n-elementową, gdzie n oznacza liczbę wierzchołków. Każdy element tej tablicy jest listą. W przypadku grafu nieskierowanego listy są dłuższe, ponieważ muszą odzwierciedlać krawędzie biegnące w obu kierunkach. Metoda ta wymaga ilości pamięci proporcjonalnej do liczby krawędzi, tak że czas potrzebny na przejrzenie całego zbioru krawędzi jest proporcjonalny do jego rozmiaru. Listy sąsiedztwa są efektywnym pamięciowo sposobem reprezentacji grafu w pamięci komputera, ponieważ zajmują pamięć rzędu **O(E)**, gdzie E oznacza liczbę krawędzi grafu. Reprezentacja listowa umożliwia efektywne przedstawienie grafów rzadkich. Wadą reprezentacji z użyciem list jest niebezpośredni dostęp do informacji o tym, czy dwa węzły są połączone krawędzią: należy przeszukać listę (zbiór) węzłów przylegających, co determinuje złożoność liniową takiej operacji.

Graf pełny jest grafem, w którym każda para wierzchołków łączy się krawędzią. Graf pełny nieskierowany o n wierzchołkach posiada  krawędzi. **Gęstością** grafu nazywamy stosunek liczby krawędzi do największej możliwej liczby krawędzi: (dla grafu nieskierowanego, w programie dwa razy mniej).

**Program** napisano w języku C++ i umieszczono w nim wyjaśniające komentarze do niektórych fragmentów kodu. Liczba wierzchołków (*V*) oraz gęstość grafu (*d*) podczas testów były odpowiednio zmieniane, co jest zaznaczone w kodzie wyróżnionym komentarzem (*//!*). Dla synchronizacji numerów wierzchołków i krawędzi z indeksami tablicy, przyjęto numerowanie ich od zera. Funkcja *graf* tworzy plik *stografow.txt*, który jest odpowiednio losowo wypełniony według specyfikacji zawartej w instrukcji, z założeniem braku węzłów izolowanych. Po wykonaniu algorytmu program zapisuje rozwiązanie do pliku *rozwiązanie.txt*.

**Celem** projektu było zaimplementowanie i przeprowadzenie testów efektywności algorytmu Dijkstry dla grafów reprezentowanych przez macierz i listę sąsiedztwa.

**Testy efektywności** przeprowadzono dla 100 grafów ważonych skierowanych dla różnej liczby węzłów: 10, 25, 50, 100, 200, 500 oraz dla różnych gęstości grafu: 25%, 50%, 75% i dla grafu pełnego. Testy przeprowadzono na tym samym sprzęcie.

1. **Opis algorytmu Dijkstry**

Najkrótsza ścieżka między dwoma wierzchołkami nie musi być tą, która składa się z najmniejszej liczby krawędzi. Algorytm Dijkstry pozwala znaleźć najkrótszą drogę (ścieżkę) w grafie ważonym do wszystkich lub do wybranego wierzchołka. Jest efektywniejszy od algorytmu Bellmanna-Forda, ale ograniczeniem jest warunek nieujemnych wag krawędzi. Opracowywany przez holenderskiego informatyka Edsgera Dijkstrę od 1956 roku i opublikowany trzy lata później. Cechuje go wydajność oraz istotny udział w rozwoju sieci komputerowych.

O rzędzie złożoności obliczeniowej algorytmu Dijkstry decyduje implementacja kolejki priorytetowej. Implementacja poprzez zwykłą tablicę (złożoność O(V2)) jest optymalna dla grafów gęstych, natomiast dla grafów rzadkich lepiej jest implementować kolejki priorytetową za pomocą kopca binarnego. Czas działania algorytmu wynosi wówczas O(E log V). Można osiągnąć czas O(E + V log V), implementując kolejkę priorytetową za pomocą kopca Fibonacciego, co pozwala wyeliminować dodatkowe (niepotrzebne) analizy mogące pojawić się w trakcie działania algorytmu.

Algorytm Dijkstry działa w następujący sposób: na początku działania ustala on odległości dla wszystkich wierzchołków w grafie na nieskończoność, z wyjątkiem wierzchołka startowego, dla którego odległość ustalana jest na 0. Głównym elementem algorytmu jest tzw. proces „relaksacji” dla każdego wierzchołka sąsiadującego z badanym. W każdym kroku algorytmu wybieramy wierzchołek *u* o najmniejszej wartości tymczasowej, przechodzimy po wychodzących z niego krawędziach do pozostałych wierzchołków tymczasowych i jeśli droga dojścia do tego wierzchołka plus waga krawędzi jest mniejsza od wartości tymczasowej tego wierzchołka, oznacza to, że algorytm znalazł „krótszą” ścieżkę. Wtedy aktualizujemy wartość dystansu oraz poprzednika tego wierzchołka. Powstaje pytanie — dla jakich wierzchołków od danego momentu działania algorytmu, poszukiwana odległość nie ulegnie już zmianie? Algorytm Dijkstry zakłada, że jest to wierzchołek o najmniejszej aktualnie wyznaczonej odległości spośród wierzchołków jeszcze nie przetworzonych. To właśnie z tego założenia wynika fakt, że algorytm Dijkstry działa prawidłowo tylko w przypadku grafów z krawędziami o nieujemnych wagach.

1. **Przebieg eksperymentu i opracowanie wyników**

Na podstawie testów efektywności opisanych we wprowadzeniu (pkt.1) policzono średni czas działania algorytmy Dijkstry dla jednego odpowiednio zaimplementowanego grafu. Wyniki przedstawiono w tabelach i na wykresach. Czas podano w milisekundach (1 ms = 10-3 s).

1. **Implementacja grafu za pomocą listy sąsiedztwa**

gęstość

l. węzłów

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 25 | 50 | 100 | 200 | 500 |
| 25% | 0,0021 | 0,0064 | 0,025 | 0,083 | 0,37 | 3,5 |
| 50% | 0,0022 | 0,0079 | 0,030 | 0,12 | 0,52 | 6,2 |
| 75% | 0,0022 | 0,0093 | 0,032 | 0,13 | 0,78 | 9,8 |
| 100% | 0,0033 | 0,0096 | 0,032 | 0,11 | 0,40 | 2,6 |

1. **Implementacja grafu za pomocą macierzy sąsiedztwa**

l. węzłów

gęstość

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 25 | 50 | 100 | 200 | 500 |
| 25% | 0,0026 | 0,0099 | 0,031 | 0,10 | 0,40 | 2,5 |
| 50% | 0,0028 | 0,012 | 0,038 | 0,13 | 0,50 | 3,2 |
| 75% | 0,0031 | 0,011 | 0,035 | 0,12 | 0,43 | 2,7 |
| 100% | 0,0032 | 0,0098 | 0,028 | 0,090 | 0,33 | 2,1 |

Na wykresach zależności czasu działania algorytmu od ilości wierzchołków przedstawiono zarówno porównanie różnych gęstości grafu dla danej struktury reprezentującej graf (pierwsze dwa wykresy), jak i porównanie struktur danych dla każdej gęstości grafu (pozostałe 4 wykresy).

1. **Podsumowanie i wnioski**

Algorytm sam w sobie jest szybki, a główny czas działania programu schodzi stworzenie 100 losowych grafów, zapisanie ich do pliku i odczytanie z pliku do odpowiedniej struktury danych (macierz sąsiedztwa lub lista sąsiedztwa). Wykresy pokazują, że implementacja grafu na macierzy pozwala algorytmowi Dijkstry działać szybciej. Należy jednak pamiętać, że w przypadku grafu nieskierowanego listy są dłuższe, gdyż muszą odzwierciedlać krawędzie biegnące w obu kierunkach. Ponadto, implementując listę w postaci kopca lub kopca Fibonacciego moglibyśmy uzyskać lepsze wyniki.

Według teorii reprezentacja listowa umożliwia efektywne przedstawienie grafów rzadkich, ponieważ gdy graf posiada małą liczbę krawędzi, to stopień efektywnego wykorzystania tablicy jest bardzo niski i większość komórek pozostanie pusta. Zajmuje ona wtedy za dużo pamięci. Z kolei kiedy graf jest gęsty lub gdy istnieje potrzeba szybkiego sprawdzenia, czy dana krawędź istnieje, wtedy korzystniejsza może okazać się reprezentacja macierzowa.

Krzywe na wykresach w przybliżeniu pokrywają się ze złożonością obliczeniową algorytmu. Czas wykonania algorytmu zwiększał się wraz ze wzrostem liczby wierzchołków. Wzrost gęstości miał większy wpływ na wyniki czasowe dla implementacji za pomocą listy niż macierzy. Otrzymane wyniki są w większości zgodne z przewidywaniami. Potwierdzają zatem teorię oraz poprawność zaimplementowania algorytmu i przeprowadzenia testów.

1. **Bibliografia**

* P. Wróblewski – „Algorytmy, struktury danych i techniki programowania”, wydanie V, Helion 2015
* T.H. Cormen, C.E. Leiserson, R.L. Rivest – „Wprowadzenie do algorytmów”, wyd. IV, WNT, Warszawa 2001
* A. Drozdek - C++. Algorytmy i struktury danych, Helion 2004
* <https://eduinf.waw.pl/inf/alg/001_search/0124.php>
* <https://eduinf.waw.pl/inf/utils/002_roz/ol011.php>
* <https://stackoverflow.com/questions/8767166/passing-a-2d-array-to-a-c-function>
* <https://miroslawzelent.pl/kurs-c++/pliki-tekstowe-zapis-odczyt-fstream/>
* <https://miroslawzelent.pl/kurs-c++/liczby-losowe-rand/>
* <https://stackoverflow.com/questions/1321137/convert-string-containing-several-numbers-into-integers>
* <https://pl.wikipedia.org/wiki/Algorytm_Dijkstry>
* <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>
* Praca magisterska Patryka Kwiatkowskiego „Implementacja podstawowej biblioteki grafów w języku C”, Politechnika Częstochowska Wydział Inżynierii mechanicznej i informatyki, promotor pracy: dr inż. Ireneusz Szcześniak, Częstochowa 2012, <http://www.irkos.org/dydaktyka/pd/kwiatkowski.pdf>
* Praca magisterska Piotra Stańczyka „Algorytmika praktyczna w konkursach informatycznych”, Uniwersytet Warszawski Wydział Matematyki, Informatyki i Mechaniki, promotor pracy: dr hab. Krzysztof Diks, Warszawa 2007, <https://www-users.mat.umk.pl/~stencel/acm/algorytmika_praktyczna.pdf>
* <https://mattomatti.com/pl/a0123?plang=cpp#elcode3>