Autor:

DAWID PICHEN (132775)

PROJEKT Z TEORETYCZNYCH PODSTAW INFORMATYKI

Temat:

ALGORYTM DIJKSTRA

Wstęp

Celem niniejszego projektu była implementacja, analiza i wskazanie złożoności obliczeniowej algorytmu Dijkstra. Algorytm Dijkstra służy do wyznaczenia najmniejszej odległości od ustalonego wierzchołku do pozostałych wierzchołków grafu skierowanego. Każda krawędź grafu ma przypisaną odległość (lub wagę, jednak jej wartość musi być dodatnia). Nazwa tego algorytmu pochodzi od nazwiska jego twórcy, holenderskiego naukowca, pioniera informatyki Edsger'a Wybe Dijkstra.

Opis programu

Analiza algorytmu Dijkstra polegała na mierzeniu czasu, jaki jest potrzebny na wyznaczenie najkrótszej ścieżki od zadanego wierzchołka do wszystkich pozostałych wierzchołków. Badałem czas wyznaczenia podzbioru elementów dla 10, 50, 75, 100, 250, 500, 750, 1000 wierzchołków grafu. Dla każdej z liczby wierzchołków, program wykonywał algorytm Dijkstra na liczbie krawędzi zależnych od liczby wierzchołków, w taki sposób, że liczba krawędzi wynosiła 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80% i 90% liczby wierzchołków. Graf był reprezentowany, jako kwadratowa macierz odległości między wierzchołkami. Każdy z elementów macierzy był liczbą całkowitą bez znaku (*unsigned int*), liczba taka jest reprezentowana w pamięci w postaci 4 bajtów (kompilator GCC dla procesora i systemu 32-bitowego). Do implementacji grafu wybrałem macierz odległości, ponieważ metoda nadaje się doskonale dla dużej liczby wierzchołków, gdyż dostęp do danych o odległościach grafu jest bardzo szybki. Oczywiście macierz kwadratowa zajmuje dużo pamięci operacyjnej, jednakże dla 1000 wierzchołków, zapotrzebowanie na pamięć wyniesie $(1000 \cdot 4B)^2$~15,3 MB. Jest to dopuszczalny blok pamięci, gdyż cały blok jeszcze zmieści się fizycznie w pamięci RAM.

Jak w poprzednich projektach, tak też w tym projekcie postanowiłem napisać program testujący algorytm Dijkstra w języku ANSI C, gdyż zależało mi na możliwie szybkiej pracy programu. Aplikacja została napisana dla systemu operacyjnego Linux, jednakże nie powinno być większych problemów z jej skompilowaniem w innych systemach UNIXowych. Program składa się z 3 plików z kodami źródłowymi i 2 plików nagłówkowych. Plik *main.c* zawiera główną pętlę programu odpowiedzialną za uruchamiania algorytmu Dijkstra dla różnej liczby wierzchołków grafu, jak i dla różnej liczby krawędzi tegoż grafu. Dla każdej pary danych o grafie (tzn. liczby wierzchołków i krawędzi) program wykonuje *L_PROB* prób wykonania algorytmu (domyślnie 10, definiowane w pliku *czas.h*). Dla każdej pary danych o grafie alokowana jest macierz grafu, zawierająca dane o odległościach wierzchołków. Odległości wybierane są losowo, ich maksymalne ograniczenie górne zdefiniowane jest poprzez definicje

OGR_DROGI pliku *graf.h* (domyślne ograniczenie 20). W programie przyjąłem, że szukam najkrótszych dróg od pierwszego wierzchołka grafu do pozostałych wierzchołków. Jeśli jednak chcielibyśmy badać najkrótsze drogi od innego wierzchołka grafu, to wystarczy zmień argument *wierzcholek_poczatkowy* funkcji *AlgorytmDijkstra()*.

Plik *graf.c* zawiera m.in. implementację algorytmu Dijkstra. Znajdują się w nim funkcje pozwalające zaalokować oraz zwolnić pamięć na macierz grafu, jak również funkcja wyświetlająca na ekranie macierz grafu, jak również zawartość wektora rozwiązań algorytmu Dijkstra.. Domyślnie program nie wyświetla macierzy grafu ani wektora wyniku, zawierającego najkrótsze drogi od wskazanego wierzchołka. Jeśli jednak chcemy zobaczyć powyższe dane, należy ustawić makrodefinicję *POKAZUJ_WSZYSTKO* w pliku *main.c*. Jeśli dodatkowo ustawimy makrodefinicję *POKAZ_KROK_PO_KROKU* w pliku graf.c, to zobaczymy każdy kolejny etap wykonywania algorytmu. W pliku tym znajduje się również funkcja wypełniająca macierz grafu wartościami losowymi, która wbrew pozorom nie jest prosta. Wypełnianie macierzy odbywa się w taki sposób, że dla każdego wierzchołka losowany jest numer wierzchołka do którego można poprowadzić krawędź, następnie sprawdzana jest wartość drogi wpisana we wskazanym miejscu w macierzy. Jeśli wartość ta jest równa nieskończoności, to znaczy, ze można w nią wpisać losową wartość drogi, tworząc tym samym nową krawędź. W przeciwnym wypadku mamy do czynienia z dwiema sytuacjami:

1) wylosowany numer wierzchołka jest taki sam jak wierzchołek z którego chcemy poprowadzić graf, lub
2) do tego wylosowanego wierzchołka krawędź została już poprowadzona.

Generowana jest taka liczba krawędzi jaka została podana w argumencie funkcji.

W pliku nagłówkowym *graf.h* oprócz nagłówków funkcji znajduje się także definicje typu strukturowego zawierających dane o elementach wektora roboczego algorytmu. Dane te, to aktualna najkrótsza droga, oraz informacja o tym, czy wierzchołek ma być usunięty dla dalszych kroków algorytmu. Znajduje się także makrodefinicja nieskończoności, bowiem liczby *unsigned int* mają ograniczone zakresy danych, dlatego przyjąłem, że nieskończoność to maksymalna wartość możliwa do zapisania na tego typu danych, czyli 4294967295.

Z kolei plik *czas.c* zawiera funkcje związane z pomiarem czasu, jak również obliczaniem wartości średniej z *L_PROB* prób wykonywania algorytmu.
Zależało mi na jak najdokładniejszym pomiarze czasu. Nie mogłem użyć funkcji *time()*, gdyż funkcja ta zwracała czas tylko z dokładnością do sekundy. Postanowiłem wykorzystać w tym celu systemową funkcję *gettimeofday(struct timeval *tp, void *tzp)*. Funkcję tę omówiłem szczegółowo w sprawozdaniach do poprzednich projektów.
Program wyświetla wyniki czasów wykonania algorytmu na ekranie w sposób szczegółowy, tzn. dla każdej próby. Tworzy on także plik *wyniki.txt* w katalogu w którym został uruchomiony. Plik ten zawiera wyniki średnich czasów realizacji algorytmu Dijkstra dla każdej zadanej ilości wierzchołków i krawędzi grafu.

Kompilacja programu

Dołączony do projektu dysk zawiera pliki źródłowe, jak również skompilowany plik wykonywalny programu. Uruchamiamy program poleceniem *projekt3* . Jeśli jednak chcemy skompilować program ze źródeł, to najpierw należy go wypakować z archiwum poleceniem:
tar -zxvf projekt3.tar.gz
następnie uruchamiamy kompilację poleceniem *make*.

Analiza złożoności obliczeniowej

Przebieg programu dla każdej pary liczby wierzchołków i krawędzi, oraz dla każdej próby przedstawia się następująco:

1) alokuj pamięć na macierzy grafu;
2) wypełnij macierz odległościami między wierzchołkami;
3) alokuj pamięć na wektor roboczy algorytmu Dijkstra;
4) wykonanie algorytmu Dijkstra;
5) wynik problemu znajduje się w wektorze roboczym;
6) zwolnij pamięć macierzy i wektora.

Analizę złożoność obliczeniową przeprowadzę dla kroku 3 i 4, gdyż zakładam, że algorytm Dijkstra znajduje wektor najkrótszych dróg dla podanej wcześniej macierzy.
Oczywistym jest, że alokacja pamięci ma złożoność obliczeniową O(1).
Pierwszym krokiem algorytmu Dijkstra jest skopiowanie zawartości wiersza, odpowiadającego wierzchołkowi początkowemu, z macierzy głównej do wektora robocza. Złożoność tego kroku wynosi O(n).
Następnie wykonywana jest pętla dopóty, dopóki w wektorze będą znajdowały się elementy nieusunięte. Pętla ta wykonuje zatem się n-1 razy.
W pierwszym kroku pętli kroku dokonywana jest relaksacja drogi, złożoność obliczeniowa tego kroku wynosi O(n).
W drugim kroku pętli kroku dokonywana znajdowany jest najmniejszy nieusunięty element wektora oraz usunięcie go z tego wektora. Złożoność obliczeniowa tego kroku wynosi O(n).
W drugim kroku pętli kroku dokonywana znajdowany jest najmniejszy nieusunięty element wektora oraz usunięcie go z tego wektora. Złożoność obliczeniowa tego kroku wynosi O(n).
Podsumowując, ogólna złożoność obliczeniowa algorytmu Dijkstra, to:
$O(1+(n-1) \cdot (n+n+n)) = O(1 + (n-1)\cdot(3n))= O(1+n^2)=O(n^2)$.
Jest to złożoność wielomianowa.

Badanie czasów sortowań

Uruchomiłem napisany program na następującym systemie komputerowym:

- procesor: Intel Pentium III 450 MHz,
- pamięć RAM: 160 MB SDRAM PC-100,
- system operacyjny: Slackware Linux 10,
- jądro systemu: 2.6.11.7.

Przed uruchomieniem programu zamknąłem wszystkie zbędne procesy, które potencjalnie mogłabym spowalniać pracę aplikacji, pozostawiłem tylko niezbędne procesy systemowe. Program został uruchomiony pod zwykłą konsolą tekstową (serwer X wyłączony). Po uruchomieniu programu uzyskałem następujące wyniki (załączone na dyskietce).

Otrzymane wyniki w tabeli:

Liczba wierzchołków	**Liczba krawędzi jako procent liczby wierzchołków**								
	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
10	0,000016	0,000017	0,000020	0,000018	0,000033	0,000020	0,000018	0,000018	0,000022
50	0,000295	0,000309	0,000327	0,000340	0,000349	0,000358	0,000347	0,000335	0,000334
75	0,000659	0,000699	0,000738	0,000765	0,000791	0,000792	0,000786	0,000767	0,000752
100	0,001171	0,001236	0,001301	0,001363	0,001387	0,001398	0,001378	0,001356	0,001332
250	0,007339	0,007719	0,008119	0,008466	0,008616	0,008613	0,008570	0,008463	0,008391
500	0,032420	0,033970	0,035378	0,036030	0,037256	0,038195	0,038008	0,037160	0,036185
750	0,073466	0,076952	0,077574	0,081713	0,085217	0,086066	0,084808	0,081546	0,078935
1000	0,129340	0,133085	0,136548	0,145897	0,151993	0,152237	0,147909	0,141655	0,137980

Wnioski:

Patrząc na dane w tabeli zauważamy, że czas wykonania algorytmu głównie zależy tylko od liczby wierzchołków. Owszem, liczba krawędzi również wpływa na otrzymane wyniki, jednak w niewielkim stopniu. Spostrzegamy następującą prawidłowość. Dla każdej liczby wierzchołków algorytm wykonuje się najdłużej, jeśli liczba krawędzi w grafie będzie odpowiadała ok. 60% liczby wierzchołków. Natomiast najlepszy czas działania algorytm uzyskuje dla jak najmniejszej liczby krawędzi w grafie. Jednak liczba krawędzi mimo wszystko ma znikomy wpływ na czas wykonania algorytmu. Jak widać z wykresu czasu działania od liczby krawędzi, wartość czasu rośnie parabolicznie wraz z wzrastającą liczbą wierzchołków. Potwierdza to badania złożoności obliczonej, która wynosi $O(n^2)$ dla tego algorytmu.

Na zakończenie warto nadmienić, że algorytm Dijkstra jest szeroko stosowany. Jest on między innymi wykorzystywany w Internetowym protokole OSPF (*Open Shortest Path First*), używanym przez zaawansowane routery do badania najlepszej trasy do innych węzłów sieci.

Wybrane wykresy