|  |  |  |  |
| --- | --- | --- | --- |
|  | **Nazwa kursu:** Struktury Danych i Złożoność Obliczeniowa (projekt nr 2.) | | |
| **Temat projektu:** „ Badanie efektywności operacji dodawania, usuwania oraz wyszukiwania elementów w różnych strukturach danych.” | | |
| **Autor:**  Piotr Ławniczak 209775 | **Data oddania:** 12.05.2015 | |
| **Grupa:** wtorek 11:15 | |
| **Prowadzący:**  Dr inż. Jarosław Mierzwa | **Ocena:** |

# Cel projektu

Celem projektu jest analiza wydajności algorytmów wyznaczających w grafie: minimalne drzewo rozpinające (algorytm Prima) oraz najkrótszą ścieżkę (algorytm Dijkstry). Polegać ona będzie na badaniach wykazujących zależność ich szybkości wykonania w zależności od rozmiaru, gęstości grafu, a także od rodzaju jego reprezentacji w pamięci komputera.

# Wstęp teoretyczny

Minimalnym drzewem rozpinającym (w skrócie ang. MST) nazywa się drzewo rozpinające danego grafu o najmniejszej z możliwych wag, tj. takie, że nie istnieje dla tego grafu inne drzewo rozpinające o mniejszej sumie wag krawędzi. Sposobów jego wyznaczania jest kilka. Jednym z nich jest algorytm Prima.

Algorytm Prima należy do grupy algorytmów zachłannych. Dysponując grafem spójnym i nieskierowanym, obliczany jest podzbiór ze zbioru krawędzi, dla którego graf nadal pozostaje spójny, jednak suma wag wszystkich takich krawędzi w owym podzbiorze jest najmniejsza z możliwych.

# Najważniejsze założenia

-wagi krawędzi są liczbami całkowitymi większymi od zera, przechowywane jako unsigned int  
-

# Reprezentacja grafów w pamięci komputera

Na potrzeby projektu zostały zaimplementowane dwa sposoby reprezentacji grafu:

1. Macierz sąsiedztwa

Sposób ten zakłada stworzenie dwuwymiarowej tablicy liczb całkowitych nieujemnych. Każdy jej element inicjalizowany jest wartością „0”, co oznacza, że między wierzchołkiem reprezentowanym przez numer wiersza a wierzchołkiem reprezentowanym przez numer kolumny nie występuje krawędź. W przypadku kiedy w danej komórce znajduje się liczba różna od zera – oznacza to występowanie krawędzi. Liczba ta jednocześnie jest jej wagą. Wymagania pamięciowe to O(V2). Operacje przejrzenia wszystkich krawędzi, sąsiadów danego wierzchołka i sprawdzenie, czy dana krawędź istnieje wykonują się w czasie O(V).

1. Lista sąsiedztwa

Reprezentacja listowa opiera się na utworzeniu w pierwszej kolejności tablicy zawierającej tyle list ile graf ma wierzchołków. Indeks tablicy odpowiada numerowi wierzchołka, a każdy element listy znajdującej się pod nim – wskazuje na jego sąsiadów (z którymi tworzy krawędź). Sąsiedzi przechowywani są w specjalnie stworzonej do tego celu strukturze, która oprócz numeru sąsiada zawiera kompletne informacje o krawędzi (wierzchołek początkowy, wierzchołek końcowy, waga). Złożoność pamięciowa wynosi O(E). Przejrzenie wszystkich krawędzi ma złożoność czasową O(E), natomiast przeglądanie sąsiadów wierzchołka i sprawdzanie istnienia krawędzi wykonuje się w czasie O(E).

# Implementacja algorytmów

1. Algorytm Prima – wyznaczanie minimalnego drzewa rozpinającego (MST)

Na początku algorytm dodaje do zbioru A reprezentującego drzewo krawędź o najmniejszej wadze, łączącą wierzchołek początkowy v z dowolnym wierzchołkiem. W każdym kolejnym kroku procedura dodaje do A najlżejszą krawędź wśród krawędzi łączących wierzchołki już odwiedzone z nieodwiedzonymi. Jeśli struktura A jest kolejką priorytetową opartą na kopcu binarnym, czasowa złożoność obliczeniowa operacji wynosi O(m \* log2n).

1. Algorytm Dijkstry – znajdowanie najkrótszych ścieżek w grafie

Algorytm Dijkstry znajduje w grafie najkrótsze ścieżki pomiędzy wybranym wierzchołkiem początkowym a wszystkimi pozostałymi, wyliczając również koszt przejścia każdej z tych ścieżek, czyli sumę wag krawędzi na ścieżce. Algorytm ma złożoność czasową O(n2) przy wykorzystaniu wyszukiwania liniowego podczas szukania wierzchołków o najmniejszym koszcie dojścia. Czas ten może być zmniejszony dzięki wykorzystaniu kolejki priorytetowej opartej na kopcu binarnym. Wtedy w korzeniu przechowywany jest wierzchołek o najmniejszej wartości kosztu dojścia. Złożoność czasowa upraszcza się do O(n \* log2n) – tyle, ile wynosi czas przywracania własności kopca.

# Plan eksperymentu

Czas potrzebny do wykonania danej operacji będzie mierzony przy użyciu licznika cykli procesora wykonanych od jego uruchomienia - TSC (Time Stamp Counter). Dostęp do niego umożliwia funkcja QueryPerformanceCounter. Jego wartość będzie odczytywana i zapisywana tuż przed rozpoczęciem mierzonego polecenia, oraz zaraz po. Różnica podzielona przez ilość cykli jakie procesor wykonuje w czasie jednej sekundy (odczytaną przy użyciu funkcji QueryPerformanceFrequency) daje szczegółowy czas w sekundach do sześciu miejsc po przecinku. Elementem struktur będzie liczba typu integer generowana losowo z zakresu [-100; 100] za pomocą funkcji *rand()*. Komputer na jakim zostaną przeprowadzone doświadczenia posiada czterordzeniowy procesor Intel i7 taktowany zegarem 3,00 GHz oraz 8 GB pamięci RAM DDR3 o częstotliwości taktowania 1600 MHz.

Kolejno wykonane zostaną:

# Wyniki eksperymentu

Na podstawie pięciu raportów uzyskanych z pięciu osobnych eksperymentów na każdej ze struktur sporządzam tabele, następnie, za ich pośrednictwem – wykresy. Wyniki dodawania oraz usuwania elementów zostały uśrednione po ich czasach ich wykonania w zależności od pozycji indeksu – początku, końca, lub miejsca losowego. W przypadku listy, celem uniknięcia zafałszowania wyników, pozycja losowa została wykluczona ze swojego udziału w uśrednianiu z powodu najprawdopodobniej błędu w implementacji, który przyczynił się do ogromnych rozbieżności czasowych. Wszystkie raporty z badań znajdują się w katalogu *raporty,* natomiast dane z nich przedstawione w postaci tabelarycznej – są zawarte także w arkuszu kalkulacyjnym *dane.xlsx*.

1. Tablica – dodawanie elementu

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| czas\ilość elem. | 2500 | 5000 | 7500 | 10000 | 12500 | 15000 | 17500 |
| początek [ms] | 0,020082 | 0,033231 | 0,043565 | 0,066305 | 0,082511 | 0,104714 | 0,113365 |
| losowo [ms] | 0,016954 | 0,026914 | 0,035941 | 0,046527 | 0,060235 | 0,073455 | 0,087012 |
| koniec [ms] | 0,018464 | 0,030746 | 0,042514 | 0,054459 | 0,069853 | 0,084149 | 0,100404 |

miejsca w pamięci

# Wnioski

Na podstawie uzyskanych wykresów nasuwają się jednoznaczne wnioski:

-Złożoność obliczeniowa zaimplementowanej przeze mnie tablicy w przypadku każdej z operacji wynosi O(n), a więc prezentuje się ona zgodnie z pierwotnymi założeniami. Jest to struktura danych o najbardziej powszechnym zastosowaniu.

-Złożoność obliczeniowa listy dwukierunkowej w przypadku dodawania i usuwania elementów na jej początek lub koniec wynosi O(1). Nietrudno zauważyć, że jest zdecydowanie szybsza niż tablica, natomiast nie zapewnia równie szybkiego dostępu do danych jak tablica.

-Kopiec binarny to idealna struktura jeśli naszym głównym kryterium wyboru jest szybkość wyszukiwania elementów. Zapotrzebowanie czasowe w przypadku wyszukiwania wraz z rozmiarem kolekcji rośnie logarytmicznie. Z uwagi na wyróżniające się na tle pozostałych struktur ułożenie danych (są one posortowane od wartości największej do najmniejszej) zapotrzebowanie czasowe na wyszukanie danego elementu zależy także od jego wartości – im większej wartości jest dany element, tym znalezienie jego zajmuje mniejszą ilość czasu. Niestety, sytuacja ma się znacznie gorzej w przypadku modyfikacji danych jakie zawiera kopiec. Mimo że złożoność algorytmu dodawania i usuwania jest identyczna jak w przypadku tablicy i listy, to na wykonanie identycznych operacji trzeba poczekać około ogromną krotność dłużej niż w przypadku listy dwukierunkowej. Winę za taką postać rzeczy ponosi najprawdopodobniej wadliwa implementacja, która po każdym pojedynczym dodaniu/usunięciu elementu przeprowadza naprawę całego kopca, analizując poprawność pozycji każdego z elementów, zamiast zająć się jedynie „wyniesieniem” świeżo dodanego elementu na odpowiednią pozycję. Miałoby to pesymistyczną złożoność O(log n), jednakże w obecnym przypadku wynosi ona O(n).

-Listy są dobrym kontenerem do przechowywania dużej ilości danych, w szczególności jeśli zachodzi częsta potrzeba ich modyfikacji. Radzą sobie one bardzo szybko z ich zapisem jak i odczytem, a ich najważniejszą cechą jest to, że nie wymagają relokacji całej kolekcji w pamięci w przypadku dodawania bądź usuwania elementów.

# Bibliografia

[1] Robert L. Kruse – „Data Structures and Program Design in C++”

[2] Piotr Wróblewski – „Struktury Danych i Techniki Programowania – Algorytmy”

[3] Thomas H. Cormen – „Wprowadzenie do Algorytmów”

[4] Wikipedia (wersja angielska) - <http://en.wikipedia.org/wiki/Binary_heap>

[5] Big-O Cheat Sheet (wersja angielska) - <http://bigocheatsheet.com/>