Politechnika Wrocławska

Projekt : Projektowanie algorytmów i metody sztucznej inteligencji Y03-51e
Grupa : Wtorek 7:30

Projekt 2
Zadania na ocene bdb (5.0):

Wykonał:

Michał Szkudlarek - nr indeksu: 259248

1 Wstęp

Zrealizowane zostało przeze mnie zadanie na ocenę bardzo dobrą. Dotyczyło ono wybrania trzech algorytmów sortowania, zaimplementowania ich w programie oraz przeprowadzenia analizy ich efektywności na podstawie podanego zbioru danych. Trzy wybrane przeze mnie algorytmy sortowania to :

- sortowanie przez scalanie (merge sort),
- sortowanie szybkie (quicksort),
- sortowanie kubełkowe (bucket sort),

Algorytmy te zostały zaprojektowane na podstawie klasy z poprzednego zadania (projekt1 - sekwencji(kolejka priorytetowa)). Pierwszym krokiem było wpisanie do struktury wszystich danych z pliku, udostępnionych do tego zadania (plik .csv). Następnie została wykonana filtracja za pomocą funkcji **DeleteInvalid()**. W kolejnym kroku został utworzony nowy obiekt struktury danych do którego zostało wpisane pierwsze dziesięć tysięcy danych z przefiltrowanej sekwencji. Na takim obiekcie zostało wykonane filtrowanie, a następnie sprawdzono przy pomocy metody **isSorted()** czy sekwencja jest poprawnie posortowana. Następnie czynność wykonano dla większej ilości danych.

Klasa wykorzystywana do przetestowania algorytmów:

```
class Sequence /* Klasa pozwalająca na dostęp dowolnych elementów
                  listy dwukierunkowej*/
{
private:
    class Node /* Węzeł listy dwukierunkowej */
    {
    public: /* Konstruktor */
        Node()
        {
            next = NULL;
            previous = NULL;
        }
        Node *getNext() { return next; }         /* Zwrócenie wskaźnika
                                                    na następny węzeł */
        Node *getPrevious() { return previous; } /* Zwrócenie wskaźnika
                                                    na poprzedni węzeł */
        ~Node()
        { /* Ustawienie następnego i poprzedniego węzła na zero */
            next = NULL;
            previous = NULL;
        }
        friend Sequence; /* Pomocnicza(zaprzyjaźniona) klasa do sequence */
    private:
        Node *next;     /* Wskaźnik na następny węzeł */
        Node *previous; /* Wskaźnik na poprzedni węzeł */
        Element elem;   /* Zawartosć węzła*/
    };
```

```
public:
    class Iterator /* Klasa umożliwiająca na poruszanie się po węzłach */
    {
    public:
        Element &operator*();                          /* Referencja do poszczególnego
                                                           elementu */
        bool operator==(const Iterator &p) const; /* Porównanie pozycji*/
        bool operator!=(const Iterator &p) const; /* Sprawdzenie czy pozycje
                                                           się różnią*/
        Iterator &operator++();                        /* Przejście do następnego
                                                           wezła*/
        Iterator &operator--();      /* Przejście do poprzedniego
                                            węzła*/
        friend Sequence;             /* Pomocnicza(zaprzyjaźniona)
                                            klasa do sequence */
    private:
        Node *wezel;        /* Wskaźnik na węzeł */
        Iterator(Node *u); /* Utworzenie od węzła, konstruktor ( u - utwórz) */
    };

public:
    Sequence();                                          /* Konstruktor*/
    Sequence(const Sequence &Sequence);                  /* Konstruktor kopiujący */
    Sequence(const Sequence &Sequence, int size);        /* Konstruktor z rozmiarem*/
    Sequence &operator=(const Sequence &Sequence);       /* Operator przyrównania
                                                             dla sekwencji*/
    int size() const;        /* Zwrócenie rozmiaru sekwencji*/
    bool isEmpty() const;    /* Sprawdzenie czy pusta sekwencja */
    Iterator begin() const; /* Zwrócenie pozycji pierwszego elementu*/
    Iterator end() const;    /* Zwrócenie pozycji ostatniego elementu*/
    void insertBack(const Element &e);  /* Umieszczenie nowego elemetnu
                                           na końcu sekwencji ( e - element) */
    void insertFront(const Element &e); /* Umieszczenie nowego elemetnu
                                           na początku sekwencji ( e - element) */
    void insert(const Iterator &p, const Element &e);/* Umieszczenie nowego elemetnu
                                                        przed iteratorem */
    void eraseFront();               /* Usunięcie pierwszego węzła sekwencji */
    void eraseBack();                /* Usunięcie ostatniego węzła sekwencji */
    void erase(const Iterator &p);       /* Usunięcie elementu o iteratorze p */
    Iterator atIndex(int i) const;       /* Zwrócenie iteratora znajdującego
                                            się na indeksie i ( i - indeks)*/
    int indexOf(const Iterator &p) const;   /* Zwrócenie indeksu iteratora p */
    float medium();          /* Mediana */
    float sumOfKeys();       /* Suma wszystkich kluczy
                                elementów sekwencji */
    float average();         /* Średnia wszystkich kluczy
                                elementów sekwencji */
    bool isSorted();         /* Sprawdzenie czy dana sekwencja
```

```
                                    jest dobrze posortowana */
    void clear();               /* Usunięcie wszystkich
                                    elementów z sekwencji */
    ~Sequence();                /* Dekonstruktor */

private:
    int numberOfItems; /* Liczba wszystkich elementów */
    Node *header;      /* Wskaźnik na pierwszy element */
    Node *trailer;     /* Wskaźnik na ostatni element */
};
```

2 Przeszukiwanie i filtrowanie elementów

Zarówno przeszukiwanie jak i filtrowanie wykorzystanej struktury wykonywane jest ze złożonością **O(n)**. Dzieje się tak dlatego, ponieważ działają one na podstawie listy dwukierunkowej, która działa z taką złożnością obliczeniową.

Filtrowanie za pomocą funkcji DeleteInvalid():

```
void DeleteInvalid(Sequence *Seq)
{
    Sequence::Iterator p(Seq->begin());
    Sequence::Iterator q = p;
    while (p != Seq->end())
    {
        if (__isnanf((*p).getKey()))
        {
            q = p;
            ++p;
            Seq->erase(q);
        }
        else
        {
            ++p;
        }
    }
}
```

Filtrowanie wykonało się w czasie 46 milisekund co pokrywa się z założoną złożonością obliczeniową.

3 Opis wybranych algorytmów sortowania

3.1 Sortowanie przez scalanie (merge sort)

Sortowanie to polega na rodzieleniu struktury danych na dwie połowy, rekurencyjne powtarzając tą czynność dla nowo powstałych podstruktur. Sortowanie w ten sposób kończy się gdy podstruktury mają jeden lub mniej elementów. W następnej kolejności są one scalane w posortowane większe struktury, aż do uzyskania posortowanej struktury początkowej. Złożoność obliczeniowa tego algorytmu to $O(n \cdot \log(n))$. Algorytm ten jest niezależy od początkowego stopnia posortowania, więc nie jest on zależny od przypadku i za każdym razem daje taki sam wynik, nie wystepują w nim przypadki "pesymistyczne" ani "optymistyczne".

3.2 Sortowanie szybkie (quicksort)

Sortowanie to polega na rozdzieleniu struktury na trzy mniejsze, z czego w jednej z nich znajdują się elementy mniejsze od wybranego wcześniej pivota(moze być to element środkowy,pierwszy, ostatni lub losowy wybany według innego schematu dostosowanego do zbioru danych), w drugiej znajdują się elementy mu równe, a w trzeciej elementy od niego większe. Algorytm ten działa ze złożonością $O(n \cdot \log(n))$ lub w pesymistyczneym przypadku ze złożonością $O(n^2)$.

3.3 Sortowanie kubełkowe (bucket sort)

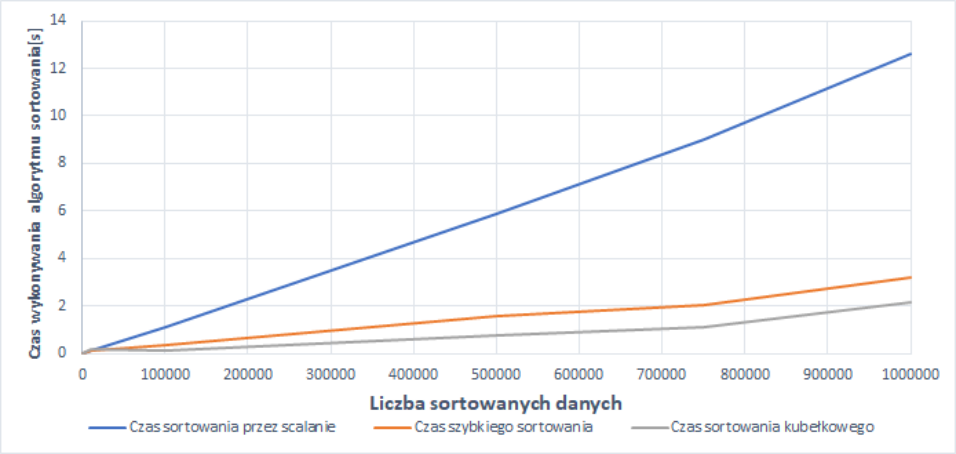
Sortowanie to polega na utworzeniu dziesięciu (lub mniejszej ilosć, zależnie od danych wejściowych)kubełków, czyli struktur danych. Natępnie należy podzielić otrzymaną wejściową strukturę danych między dziesięć kubełków patrząc na nich wartość - do pierwszego trafią elementy w przedziale od minimum do $0,1 \cdot (maksimum - minimum)$. W ten sposób otrzymamy 10 lub mniej struktur, które następneie należy posortować innym algorytmem (w tym przypadku zostało wykorzystane sortowanie szybkie), a następnie scalić, zaczynając od kubełka zawierającego najmniejsze lementy. Algorytm ten może mieć złożoność taką samą jak sortowanie szybkie, gdyż dodatkowe operacje wykonywane są ze złożonością maksymalnie $O(n)$.

4 Opis przeprowadzonej analizy efektywności algorytmów

Wykonano sortowanie na każdym z algorytmów na strukturach zawierających kolejno: **10 000, 100 000, 500 000, 750 000** oraz maksymalną liczbę danych z pliku. Dane te zostały zaprezentowane w tabeli oraz wykresie poniżej:

ilość danych	merge sort[s]	quicksort[s]	bucket sort[s]	mediana	średnia artmetyczna
10 000	0,101	0,142	0,155	5	5,46
100 000	1,078	0,351	0,141	7	6,09
500 000	5,896	1,562	0,759	7	6,67
750 000	8,986	2,031	1,121	7	6,67
max	12,629	3,201	2,137	7	6,64

Tabela 1: Tabela przedstawiająca wyniki pomiarów wykonanych podczas analizy



Wykres 1: Wykres przedstawiający wyniki pomniarów

Wykres 2: Wykres słupkowy przedstawiający wyniki pomniarów

5 Wnioski oraz podsumowanie

- Z otrzymanych danych możemy zauważyć, że najwięcej czasu algorytmowi zajęło sortowanie przez scalanie. Może być to spowodowane faktem, że przypadek wejściowej struktury danych jest bliższy pozytywnego przypadku dla szybkiego sortowania co wpływa na czas sortowania zarówno szybkiego jak i kubełkowego, natomiast sortowanie przez scalanie nie jest zależne od początkowego ułożenia danych.

- Powodem takich wyników czasu może być również fakt, że bardzo duża ilość elementów może się powtarzać, więc sortowanie szybkie tworząc stukturę danych o tej samej wartosci nie musi powtarzac tej czynnosci dla tych samych danych, gdyz znajduja sie one w sekwencji dla danych równych pivotowi.

- Co również można zauważyć jak i wywnioskować najlepiej w tym przypadku ssprawdzi się sortowanie kubełkowe, ponieważ nie musi ono rozpatrywać około miliona elemetnów w złożoności $O(n \cdot \log(n))$, a korzysta ona z kilku mniejszych struktur, które łatwiej posortować wspomajac się szybkim sortowaniem.

6 Bibliografia

Literatura

[1] Wikipedia. Sortowanie szybkie.
url: `https://pl.wikipedia.org/wiki/Sortowanie_szybkie`

[2] Wikipedia. Sortowanie przez scalanie.
url: `https://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie`

[3] Wikipedia. Sortowanie kubełkowe.
url: `https://pl.wikipedia.org/wiki/Sortowanie_kube\OT4\lkowe`

[4] Informacje z wykładu prowadzonego przez dr inż. Łukasza Jelenia

[5] Sorting Algorithms Posted by Leonardo Galler and Matteo Kimura
url: `https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/`

[6] Sorting Algorithms - GeeksforGeeks
url: `https://www.geeksforgeeks.org/sorting-algorithms/`