Projektowanie Algorytmów i Metody Sztucznej Inteligencji Projekt 1

Artur Ziółkowski

259276

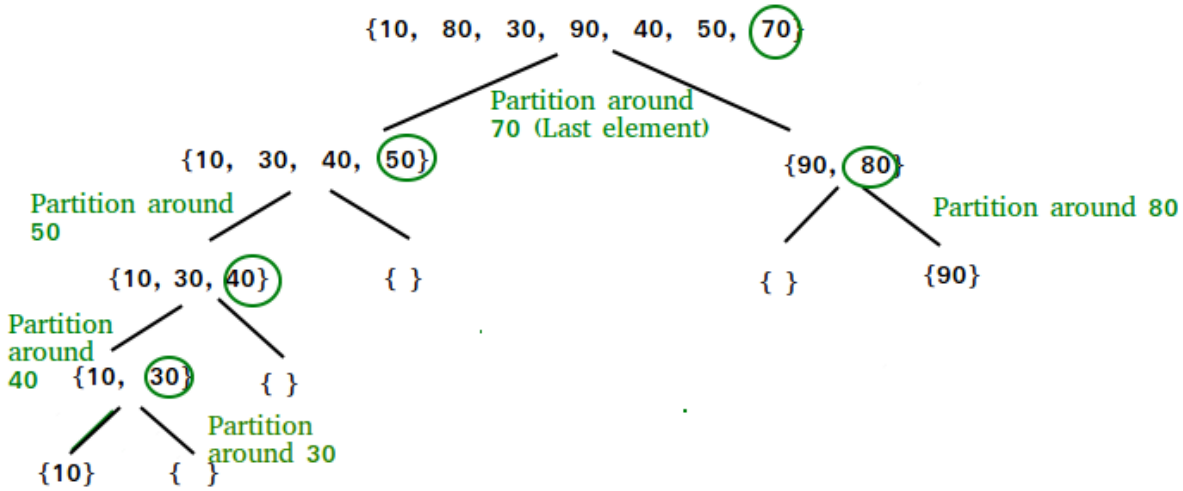
Wstęp

Celem projektu było zaimplementowanie wybranych algorytmów sortowania, a następnie sprawdzenie wydajności ich działania. Miało to służyć porównaniu tych algorytmów w różnych warunkach. Cały projekt został wykonany w języku c++ zgodnie ze standardami C++17. Dostęp do całego projektu można znaleźć w poniższym repozytorium. W projekcie wykorzystano zewnętrzną bibliotekę „google tests".

https://github.com/ArturZiolkowski1999/SortAlgorithms.git

Algorytm quick sort

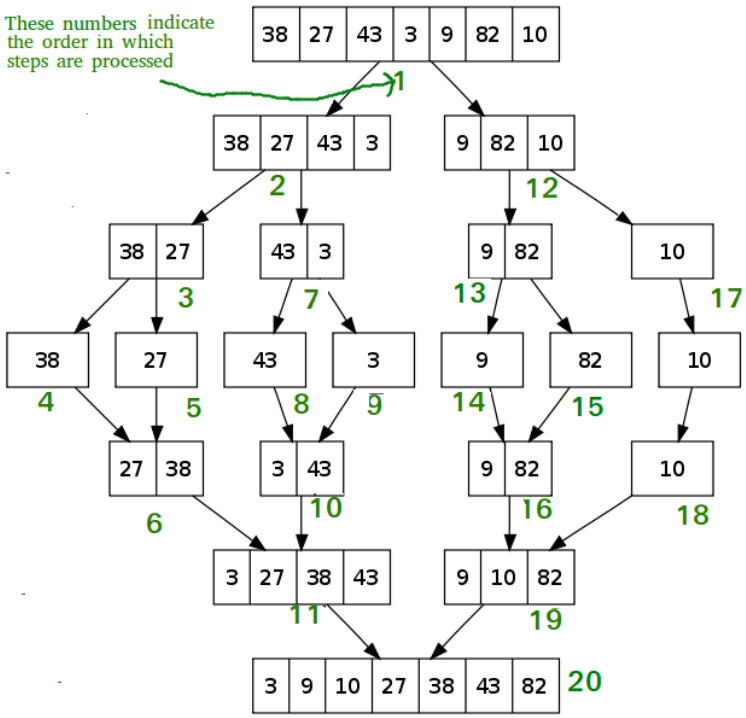
„Quick sort", jest algorytmem rekurencyjnym. Na początku wybierany jest element zwany pivotem (w mojej implementacji, jest to zawsze środkowy element sortowanej tablicy). Po tym wyborze tablica dzielona jest na dwie. Jedna składa się z elementów większych od pivota, a druga z mniejszych. Nowo powstałe tablice rekurencyjnie poddaje się tej samej procedurze. Kiedy wszystkie zostaną podzielone na najmniejsze możliwe elementy, pivoty z poprzednich wykonań rekurencji, wraz z pozostałymi elementami łączone są w jedną, posortowaną tablicę. Dobrze ilustruje to poniższy rysunek:



Algorytm ten w najgorszym wypadku przyjmuje złożoność obliczeniową O(n^2), a w najgorszym O(n). W rzeczywistości jednak takie przypadki są skrajnie nieprawdopodobne, dla większych tablic. Średnia złożoność obliczeniowa algorytmu wynosi O(nlogn). Ten algorytm dzięki zastosowanej pętli zamieniającej wartości w tablicy działa szybciej niż inne wydajne algorytmy, o ile ilość danych nie jest dostatecznie duża. Dziej się tak ponieważ dla większości komputerowych architektur pętla działa szybciej niż wyłącznie wywoływanie rekurencyjne.

Algorytm merge sort

„Merge sort”, podobnie jak quick sort, jest algorytmem rekurencyjnym. Na początku tablica jest rekurencyjnie dzielona na pojedyncze elementy. Następnie elementy tablicy są łącz one, tak aby utworzyły nowy większy posortowany element. Proces ten się powtarza, aż do uzyskania całej posortowanej tablicy. Dobrze pokazano to na poniższym rysunku:



Złożoność obliczeniowa tego algorytmu bez względu na okoliczności zawsze przyjmuje O(nlogn), spowodowane jest to tym, że liczba wywołań rekurencyjnych zależy wyłącznie od ilości elementów tablicy. Ze względu na swoją stałą i stosunkowo niską złożoność obliczeniową ten algorytm jest dobrym wyborem dla dużych ilości danych, oraz wtedy kiedy zależny nam na stałym czasie wykonania operacji. Ze względu na to, że algorytm, w przeciwieństwie do „quick sort", nie polega na działaniu pętli, dla większości architektur komputerowych będzie on działał wolniej. W związku z tym nadaje się on świetnie dla innych niż tablica obiektów przechowujących dane, które nie udostępniają szybkiego indeksowania (np. lista). Wadą tego algorytmu jest to, że potrzebuje dodatkowego miejsca na przechowywanie tymczasowych struktur danych.

Algorytm heap sort

Algorytm „heap sort" swoją nazwę i zasadę działania wziął ze struktury danych drzewa binarnego. Na początku wybierany jest jeden element, tak zwany „root", który posiada dwie gałęzie, każda ze swoją wartością. Jeśli lewa lub prawa wartość na gałęzi jest większa o root-a, to największa z nich go zastępuje. Następnie rekurencyjnie proces jest powtarzany, aż w wyniku powstanie drzewo binarne, które na górze ma najwyższą wartość. Wartość z góry wybierana zamieniana jest z ostatnim nieposortowanym elementem tablicy, a cały proces zamiany miejsc root-a i gałęzi jest powtarzany, aż powstanie posortowana tablica jako ostatni element. Poniższy link przedstawia animację, która pokazuje działanie tego algorytmu w przejrzysty sposób.

https://www.youtube.com/watch?v=MtQL_ll5KhQ&ab_channel=GeeksforGeeks

Złożoność tego algorytmu wynosi O(nlogn), co czyni go efektywnym. W przeciwieństwie do merge sort, nie wymaga on dodatkowej pamięci potrzebnej do zapamiętania tymczasowych struktur danych, co czyni go dobrą alternatywą dla tej metody.

Algorytm insertion sort

Idea stojąca za tym algorytmem jest naturalna dla człowieka. Z tablicy wybierane są po kolei elementy a następnie ustawiane są one w odpowiednie miejsca w innej tablicy. Aż elementy pierwszej tablicy się skończą i utworzą posortowaną tablicę drugą. Problemem takiej metody jest to, że częste wstawianie elementu w konkretne miejsce tablicy jest bardzo kosztowne obliczeniowo. Algorytm ten nie będzie działał dobrze, gdy takich operacji będzie sporo. Nie korzysta on jednak z rekurencji, dzięki czemu dla bardzo małych tablic będzie on działał stosunkowo szybko. Złożoność obliczeniowa tego algorytmu w najgorszym przypadku wynosi O(n^2), co jest również złożonością obliczeniową przypadku średniego.

Rozwiązania techniczne

W celu wygodnego i szybkiego testowania stworzona została klasa szablonowa „SortableList”, posiada ona tablicę o określonych przez programistę wymiarach (zaimplementowaną za pomocą std::array, w celu zwiększenia czytelności kodu). Ponad to posiada również opisane wyżej metody sortowania oraz funkcję do wylosowania wartości dla określonego zakresu tablicy.

```
template <typename T, unsigned int dimension>
class SortableList {
private:
    std::array<T, dimension> sortable_array;
    void merge(int const left, int const mid, int const right);
    void heapify(int const size, int const root);
    void intro_sort_utility(int const begin, int const end, int const depth_limit);
public:
    SortableList();
    void reshuffle(float percentage = 0);
    void quick_sort(int const begin = 0, int const end = int(dimension-1));
    void merge_sort(int const begin = 0, int const end = int(dimension-1));
    void insertion_sort(int const begin = 0, int const end = int(dimension-1));
    void heap_sort(int const size = int(dimension));

    template<typename T1, unsigned int dimension1>
    friend std::ostream &operator<<(std::ostream &ost, SortableList<T1, dimension1> &srt_list);
    void set_array(std::array<T, dimension> &srt_array);
};
```

Do testowania czasu działania konkretnej metody stworzono klasę wykorzystującą metody z biblioteki <chrono>.

```
class Benchmark
{
    private:
        std::chrono::time_point<std::chrono::high_resolution_clock> start_point;
        float duration; // [us]

    public:
        Benchmark(){
            this->start_point = std::chrono::high_resolution_clock::now();
            this->duration = 0;
        }

        ~Benchmark(){
            Stop();
        }

        float Stop(){
            auto end_point = std::chrono::high_resolution_clock::now();

            auto start = std::chrono::time_point_cast<std::chrono::microseconds>(this->start_point).time_since_epoch().count();
            auto end = std::chrono::time_point_cast<std::chrono::microseconds>(end_point).time_since_epoch().count();
            this->duration += (end - start);
            return this->duration;
        }

        float get_duration(){
            return this->duration;
        }

        void Start(){
            this->start_point = std::chrono::high_resolution_clock::now();
        }

};
```

Pojedynczy test wydajności wygląda następująco:

```
{
int iterations = 100;
float time_reasult_quick_sort, time_reasult_heap_sort, time_reasult_merge_sort;
SortableList<char, 1000000> *srt_lst = new SortableList<char, 1000000>();
test_sort_array_many_times(*srt_lst, 50);
delete srt_lst;
srt_lst = NULL;
}
```

Do działania wykorzystuje on funkcję test_sort_array_many_times, pokazaną poniżej.

```cpp
template <typename T, unsigned int dimension>
std::tuple<float,float,float> test_sort_array_many_times(SortableList<T, dimension> srt_lst, float percentage = 0, int iterations = 100){
    Benchmark timer_quick_sort;
    Benchmark timer_merge_sort;
    Benchmark timer_heap_sort;

    for(int i = 0; i < iterations; ++i){
        // quick sort
        srt_lst.reshuffle(percentage);
        timer_quick_sort.Start();
        srt_lst.quick_sort();
        timer_quick_sort.Stop();
        // heap sort
        srt_lst.reshuffle(percentage);
        timer_heap_sort.Start();
        srt_lst.heap_sort();
        timer_heap_sort.Stop();
        // merge sort
        srt_lst.reshuffle(percentage);
        timer_merge_sort.Start();
        srt_lst.merge_sort();
        timer_merge_sort.Stop();
    }

    float mean_time_reasult_quick_sort = timer_quick_sort.get_duration()/iterations;
    float mean_time_reasult_heap_sort = timer_heap_sort.get_duration()/iterations;
    float mean_time_reasult_merge_sort = timer_merge_sort.get_duration()/iterations;

    std::cout<< percentage<<" proc sorted "<< dimension <<" elements:"<<std::endl;
    std::cout<<"quick sort time:"<<std::endl;
    std::cout<<mean_time_reasult_quick_sort<< "[us]" <<std::endl;
    std::cout<<"heap sort time:"<<std::endl;
    std::cout<<mean_time_reasult_heap_sort<< "[us]" <<std::endl;
    std::cout<<"merge sort time:"<<std::endl;
    std::cout<<mean_time_reasult_merge_sort<< "[us]" <<std::endl;
    return std::make_tuple(timer_quick_sort.get_duration(), timer_heap_sort.get_duration(), timer_merge_sort.get_duration());;
```

Pliki nagłówkowe znajdują się w folderze „inc”, pliki źródłowe „src”, testy jednostkowe z zewnętrznej biblioteki „google tests” w folderze „tst”.

Wyniki

Otrzymane wyniki dla różnych rozmiarów tablic, różnego poziomu ich uporządkowania, dla różnych metod przedstawiono w tabeli poniżej.

Elements:		Quicksort [us]	Merge Sort	Heap Sort	Insertion Sort
1 000 000					
	0% posortowanych	232090	284992	647229	-
	25% posortowanych	234008	272855	641095	-
	49% posortowanych	-	-	-	-
	50% posortowanych	256844	258418	635413	-

	75% posortowanych	207620	245691	638693	-
	95% posortowanych	189671	235776	657858	-
	99% posortowanych	177434	234366	641331	-
	99 posortowanych	173489	235472	604852	-
	100% posortowanych	163062	235123	567961	-
	posortowany odwrotnie	167206	230581	602369	-
500 000					
	0% sorted	112695	139187	304961	-
	25% sorted	110922	134094	303606	-
	49% sorted	-	-	-	-
	50% sorted	127711	126280	300173	-
	75% sorted	98205	118950	301828	-
	95%	89460	113323	306176	-
	99%	84167	113928	307829	-
	99,70%	80868	112228	285506	-
	100%	75166	110997	267249	-
	reverse 100%	79845	112808	289413	-
100 000					
	0% sorted	20592	25965	54209	-
	25% sorted	20167	24511	53901	-
	49% sorted	-	-	-	-
	50% sorted	24288	23160	53789	-
	75% sorted	17422	21670	53165	-
	95%	15969	20879	54450	-
	99%	14584	20536	53529	-
	99,70%	13624	20043	49439	-
	100%	13066	20045	46877	-
	reverse 100%	13746	20198	49022	-
50 000					
	0% sorted	9560	12185	25119	-
	25% sorted	9480	11619	25258	-
	49% sorted	9094	10977	25378	-
	50% sorted	11717	10961	25069	-
	75% sorted	8161	10325	24928	-
	95%	7354	9807	25326	-

	99%	6724	9749	25154	-
	99,70%	6433	9788	23959	-
	100%	6249	9902	22991	-
	reverse 100%	6543	9887	23788	-
10 000					
	0% sorted	1710	2257	4402	267651
	25% sorted	1677	2139	4411	251140
	49% sorted	1631	2031	4428	204056
	50% sorted	2124	2025	4380	202116
	75% sorted	1446	1892	4342	118220
	95%	1278	1770	4375	26184
	99%	1155	1770	4400	5527
	99,70%	1069	1747	4222	1727
	100%	1037	1753	3979	115
	reverse 100%	1098	1783	3948	536683

Porównanie metod

Na podstawie uzyskanych wyników można stwierdzić, że quick sort jest najbardziej uniwersalną metodą sortowania tablic, spośród tych które były testowane. W niewielkim stopniu przyspiesza, dla częściowo posortowanych tablic. Widoczna jest jednak anomalia, dla poziomu posortowania równego 50%. Wynika ona z tego, że szybkość sortowania zależy bardzo od wybrania odpowiedniego pivotu. W mojej implementacji pierwszy pivot zawsze był środkowym elementem tablicy, w związku z czym, gdy przesortowano dokładnie 50% tablicy pivot był nad wymiarowo duży, co powodowało spowolniony czas sortowania dla tego algorytmu. Przeprowadzono eksperyment, w którym przesortowano 49 % tablicy, i nie zaobserwowano takiej zależności. Dowidzi to słuszności mojej hipotezy. Dla bardzo dużych tablic różnica szybkości pomiędzy quick sortem, a merge sortem staje się coraz mniej widoczna. Sugeruje to, że dla dużych tablic skuteczniejszą i bardziej stabilną metodą może być merge sort. Heap sort okazał się być wolniejszy ponad dwukrotnie od merge sort, dla każdego rozmiaru tablicy, co wskazuje na to, że metoda ta, pomimo prostszej implementacji, nie jest konkurencyjna. Ciekawe wyniki uzyskano dl algorytmu insertion sort. Dla tego algorytmu szybkość sortowania mocno zależała od ilości elementów i stopnia posortowania. Dla małej ilości danych do posortowania algorytm ten był najskuteczniejszy, a dla dużej skrajnie niewydajny. Był tak nie wydajny, że zaniechano pomiary czasu sortowania dla tego algorytmu dla większych tablic. Ograniczyło to zużycie prądu, które mogłoby nadszarpnąć budżet studenta sponsorującego doświadczenia.

Źródła

Kod źródłowy:

https://github.com/ArturZiolkowski1999/SortAlgorithms.git

Opracowane strony:

https://www.youtube.com/watch?v=oEx5vGNFrLk&ab_channel=TheCherno

https://www.geeksforgeeks.org/quick-sort/

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/heap-sort/

https://www.geeksforgeeks.org/insertion-sort/

https://www.youtube.com/watch?v=MtQL_ll5KhQ&ab_channel=GeeksforGeeks