Algorithmen zur Bestimmung des Medians

Hannes Höttinger und Josua Kucher

FH Technikum Wien, Game Engineering und Simulation, Wien, AUT

1 Aufgabenstellung

Ziel ist der Performancevergleich verschiedener Varianten der Mediansuche von erzeugten Zufallszahlen. Folgende 4 Algorithmen wurden in C++ implementiert:

1. Quicksort mit Randomized-Pivot und Ausgabe des mittleren Elements
2. Randomized-Select rekursiv
3. Median of Medians Algorithmus
4. C++ STL nth_element

2 Quicksort

ist ein rekursiver, nicht-stabiler Sortieralgorithmus, der nach dem divide and conquer Prinzip vorgeht. Die zu sortierende List wird in zwei Teillisten getrennt. Die Stelle an der die Liste geteilt wird nennt sich das Pivotelement. Danach wird die Liste so sortiert, dass alle Elemente die kleiner als das Pivotelement sind auf die linke Seite kommen, alle die größer sind auf die rechte Seite. Danach werden die Teillisten in sich selbst sortiert (ein Aufruf für die linke Seite + ein Aufruf für die rechte Seite). Dies erfolgt solange bis die Liste vollständig sortiert ist. Die Wahl des Pivotelements ist bei Quicksort für die Laufzeit sehr entscheidend. Implementiert wurde ein Random-Partitioning, da der Code für den Randomized-Select Algorithmus wiederverwendet werden kann. Es ergibt sich folgende $\mathcal{O}$-Notation:

$$Worstcase : \mathcal{O}(n^2) \tag{1}$$

$$Averagecase : \mathcal{O}(n \cdot \log(n)) \tag{2}$$

3 Randomized-Select Recursive

Im Gegensatz zu Quicksort, welches an beiden Seiten die geteilte Liste gekursiv verarbeitet, arbeitet Randomized-Select jeweils nur auf einer Seite. Der Unterschied zeigt sich bei der Analyse der erwarteten Laufzeit. In dem Code-Beispiel 1 wird der Randomized-Select Algorithmus beschrieben. Der Code liefert das

i-kleinste Element der Liste A. In der ersten Zeile überprüft der Algorithmus ob die Liste nur aus einem Element besteht und geben diesen Wert zurück. Wenn nicht dann rufe die Funktion RANDOMIZED-PARTITION auf. Hier wird ein zufälliges Pivotelement gewählt und die Liste wie bei Quicksort zweigeteilt. A[q] ist das Pivotelement. In Zeile 4 wird k berechnet, welches die Anzahl der Elemente der linken Hälfte entspricht, plus ein Wert für das Pivotelement. Entspricht dies dem i-kleinsten Element, dann liefert die Funktion den return Wert. Wenn nicht wird überprüft in welchem Teil das i-kleinste Elemente der beiden Teillisten liegt und ruft rekursiv diesen Teil auf, solange bis der i-kleinste Wert gefunden wurde.

Listing 1: Randomized-Select Pseudocode von Thomas H. Cormen (2009)

```
RANDOMIZED-SELECT(A, p, r, i)
1 if p == r
2 return A[p]
3 q = RANDOMIZED-PARTITION(A, p, r)
4 k = q - p + 1
5 if i == k // the pivot value is the answer
6 return A[q]
7 elseif i < k
8 return RANDOMIZED-SELECT(A, p, q - 1, i)
9 else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Die worstcase Laufzeit beschreibt Thomas H. Cormen (2009) mit $\mathcal{O}(n^2)$, denn es könnte passieren, dass wenn man das kleinste Element sucht, die Partitionierung immer über das größte Element erfolgt. Es ergeben sich aber aus der Randomisierung folgende Werte:

$$Worstcase : \mathcal{O}(n^2) \tag{3}$$
$$Averagecase : \mathcal{O}(n) \tag{4}$$

4 Median of Medians

Der Median der Mediane geht nach dem divide and conquer Prinzip vor und basiert auf dem Quickselect Algorithmus. Die Datenmenge wird in Buckets mit der Größe 5 unterteilt. Diese werden sortiert und anschließend das mittlere Element als Median für den jeweiligen Buckets ausgegeben. Diese Mediane werden schließend wieder in Buckets der Größe 5 unterteilt und so weiter. Der daraus resultierende approximierte Median wird als Pivotelement für den Quickselect Algorithmus verwendet um den echten Median zu bestimmen.

Dieser Algorithmus besitzt eine $\mathcal{O}(n)$-Notation von:

$$Wortcase : \mathcal{O}(n) \tag{5}$$
$$Bestcase : \mathcal{O}(n) \tag{6}$$

Daraus ergibt sich folgende Datenstruktur:

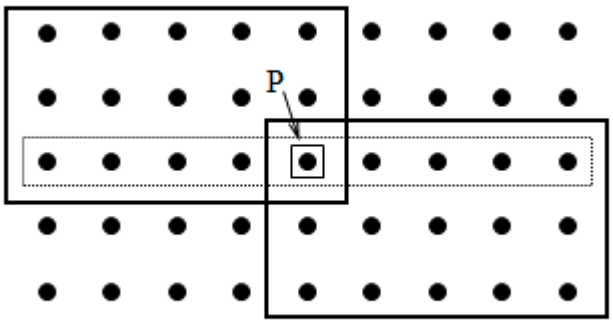

Abbildung 1: Median der Mediane

Abb. 1 zeigt den Median der Mediane, hier markiert mit P, und die Zwischenmedian markiert mit der horizontalen gestrichelten Box.

5 Testprotokoll

Zur Durchführung des Tests wurden eine Reihe an Zufallszahlen mit Hilfe des Xorshift Algorithmus generiert. Diese Zahlen wurden in vier Arrays geschrieben, die jeweils einem der Algorithmen zuzuordnen sind. Die Arrays sind untereinander identisch, um einen späteren Vergleich der Mediane durchführen zu können. Für jeden Testdurchlauf werden diese Zahlen neu generiert und zugewiesen. Für den Fall, dass die Anzahl der zufallsgenerierten Elemente gerade ist, wird in allen Algorithmen immer der untere Median verwendet. Gesamt wurden **zehn** Testdurchläufe durchgeführt die folgendermaßen aufgebaut sind:

1. Generieren der Zufallszahlen
2. nth-Element Median
3. Quicksort Random-Pivot Median
4. Randomized Select Median
5. Median of Medians

Für die Berechnung der Testzeiten der Algorithmen wurde eine eigene Klasse implementiert (siehe Anhang A für die verwendete Implementierung).

6 Ergebnisse

Die Tab. 1 zeigt die gemittelten Zeiten für den Testdurchlauf (Durchläufe = 10). Es wurden $2 \cdot 10^6$ Zufallszahlen mit Xorshift erzeugt. Das Ergebnis zeigt, dass der randomisierte Algorithmus wesentlich schneller ist als Quicksort und auch deutlich schneller als Median of Medians, obwohl dieser eine fixe $\mathcal{O}$-Notation von $\mathcal{O}(n)$ hat. Bei Programmausführung werden die erstellten Zufallszahlen pro Testdurchlauf extra in einem .txt File gespeichert. Weiters wird die durchschnittliche Laufzeit der Algorithmen in einem weiteren .txt File gespeichert (*testresults.txt*). Das Format ist wie folgt: **hh:mm:ss.ms**

Algorithmus	Average Running Time [s] for n = $2 \cdot 10^6$
nth-Element	0.025
Quicksort	0.288
Randomized-Select	0.023
Median of Medians	0.376

Tabelle 1: Ergebnis Median Frequenzen

Literatur

Thomas H. Cormen, Charles E. Leiserson, R. L. R. C. S. (2009). *Introduction to Algorithms* . MIT, 3rd edition edition.

A Zeitmessung

Listing 2: Timer.h

```
#pragma once
#include <vector>

struct time
{
    int hours = 0;
    int minutes = 0;
    int seconds = 0;
    int milliseconds= 0;

    time operator+(time rhs)
    {
        this->hours += rhs.hours;
        this->minutes += rhs.minutes;
        this->seconds += rhs.seconds;
        this->milliseconds += rhs.milliseconds;
        return (*this);
    }
    time operator/(int factor)
    {
        this->hours = this->hours/factor;
        this->minutes = this->minutes / factor;
        this->seconds = this->seconds / factor;
        this->milliseconds = this->milliseconds / factor;
        return (*this);
    }
};
class TimerFunc {
public:
    TimerFunc();
    ~TimerFunc();
    void startTimer();
    void stopTimer(char);
    void saveTimes(const char*);
private:
    time times;
    time calculateTimeAverage(std::vector<time> times);
    LARGE_INTEGER frequency;        // ticks per second
    LARGE_INTEGER t1, t2;           // ticks
    double elapsedTime;
    std::vector<time>  Timesnth;
    std::vector<time>  Timesqs;
    std::vector<time>  Timesrand;
    std::vector<time>  Timesmedian;
};
```

Listing 3: Timer.cpp

```
#pragma once
#include "stdafx.h"
#include "Timer.h"
#include <stdio.h>

TimerFunc::TimerFunc()
{
}

TimerFunc::~TimerFunc()
{
}

void TimerFunc::startTimer()
{
    // get ticks per second
    QueryPerformanceFrequency(&frequency);

    // start timer
    QueryPerformanceCounter(&t1);
}

void TimerFunc::stopTimer(char select)
{
    // stop timer
    QueryPerformanceCounter(&t2);

    // compute the elapsed time
    elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / ↩
        frequency.QuadPart;

    int milliseconds = (int)(elapsedTime) % 1000;
    int seconds = (int)(elapsedTime / 1000) % 60;
    int minutes = (int)(elapsedTime / (1000 * 60)) % 60;
    int hours = (int)(elapsedTime / (1000 * 60 * 60)) % 24;

    printf("%.2d:%.2d:%.2d.%.3d; ", hours, minutes, seconds↩
        , milliseconds);

    time temp;
    temp.hours = hours;
    temp.minutes = minutes;
    temp.seconds = seconds;
    temp.milliseconds = milliseconds;

    switch (select)
    {
        case 'n':
            Timesnth.push_back(temp);
            break;
        case 'r':
            Timesrand.push_back(temp);
            break;
```

```
        case 'm':
            Timesmedian.push_back(temp);
            break;
        case 'q':
            Timesqs.push_back(temp);
            break;
    }
    elapsedTime = 0;
}
void TimerFunc::saveTimes(const char* filename)
{
    using namespace std;

    time avergaer;
    time avergaen;
    time avergaeq;
    time avergaem;

    avergaem = calculateTimeAverage(Timesmedian);
    avergaen = calculateTimeAverage(Timesnth);
    avergaeq = calculateTimeAverage(Timesqs);
    avergaer = calculateTimeAverage(Timesrand);

    FILE * pFile;

    fopen_s(&pFile,filename,"w");
    fprintf(pFile, "Average nth-Element: %.2d:%.2d:%.2d.%.3↩
        d\n", avergaen.hours, avergaen.minutes, avergaen.↩
        seconds, avergaen.milliseconds);
    fprintf(pFile, "Average Quicksort: %.2d:%.2d:%.2d.%.3d\↩
        n", avergaeq.hours, avergaeq.minutes, avergaeq.↩
        seconds, avergaeq.milliseconds);
    fprintf(pFile, "Average random Pivot: %.2d:%.2d:%.2d↩
        .%.3d\n", avergaer.hours, avergaer.minutes, ↩
        avergaer.seconds, avergaer.milliseconds);
    fprintf(pFile, "Average Median of Medians: %.2d:%.2d↩
        :%.2d.%.3d\n ", avergaem.hours, avergaem.minutes, ↩
        avergaem.seconds, avergaem.milliseconds);
    fclose(pFile);
}
time TimerFunc::calculateTimeAverage(std::vector<time> ↩
    times)
{
    time temp;
    for (time n : times)
    {
        temp = temp + n;
    }
    return temp / times.size();
}
```