

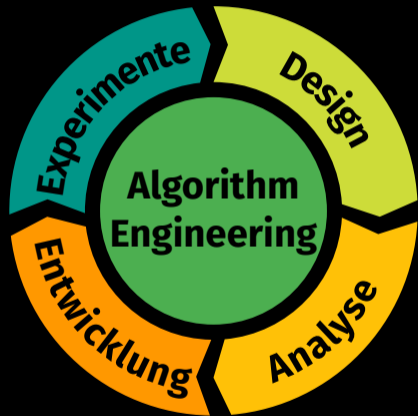
ALGORITHM ENGINEERING: BIT-VEKTOREN UND RANK-ANFRAGEN

BWINF-WORKSHOP DER TU DORTMUND 2020

Florian Kurpicz



Algorithm Engineering



Design

- ▶ Lösung für das Problem überlegen

Analyse

- ▶ Theoretische Analyse der Idee

Entwicklung

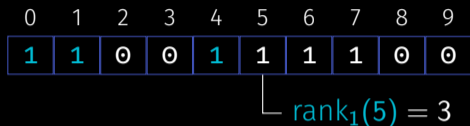
- ▶ Praktische Umsetzung der Lösung

Experimente

- ▶ Implementierung auswerten
- ▶ Ergebnisse für besseres Design nutzen

Das Problem

1. Markiere eine Teilmenge von (geordneten) Elementen und (**Bit-Vektor**)



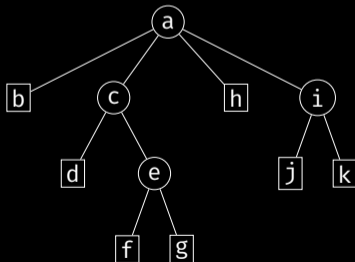
2. finde heraus, wieviele kleinere Elemente (nicht) markiert sind. (**Rank-Anfragen**)

Wofür kann man Bit-Vektoren noch verwenden?

- ▶ Kompakte Darstellung von Bäumen
(mit weiteren Anfrage-Arten)

ab ch id ejkfg
LOUDS 111100110011001100000

- ▶ ...



Design, Analyse, Entwicklung & Experimente: Simpel Bit-Vektoren

std::vector<char/int/...>

- ▶ Einfacher Zugriff auf Bits
- ▶ Sehr groß 1/4/... Bytes für ein Bit

Entwicklung und **Experimente**

Live Demo

std::vector<bool>

- ▶ Funktioniert in C++ (1 Bit je Bit)
- ▶ Einfacher Zugriff auf Bits
- ▶ Umsetzung implementierungsabhängig

Design und Analyse: Performanter Bit-Vector

`std::vector<uint64_t>`

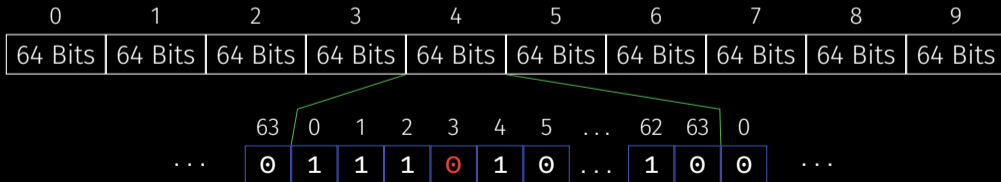
- ▶ Benötigt auch 8 Bytes für ein Bit?
- ▶ Speicher 64 Bits in jedem Eintrag!
- ▶ Wie funktioniert der Zugriff auf einzelne Bits?

Zugriff auf das i -te Bit

1. $i/64$ gibt die Position im Vector an
2. $i\%64$ gibt das Bit im Wort an

Zugriff auf das 259-te Bit:

$$259/64 = 4 \text{ und } 259\%64 = 3$$



Entwicklung: Wie kommen wir jetzt an das Bit?

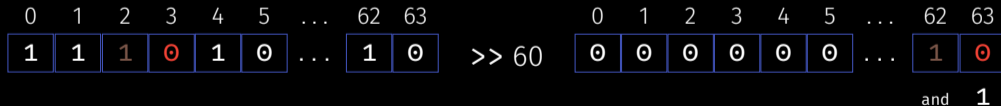
```
// Wir haben einen Bit-Vector  
std::vector<uint64_t> bit_vector;
```

```
// Zugriff auf das i-te Bit  
uint64_t block = bit_vector[i/64];  
bool bit = (block >> ( 63 - (i % 64)) ) & 1ULL;
```

Bits nach rechts verschieben

um

verunde mit 1



Entwicklung: Geht das besser?

`(block >> (63-(i%64))) & 1ULL;`

- ▶ Fülle Bit-Vector von links nach rechts

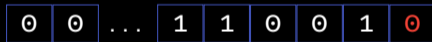


- ▶ Assembler-Code:

```
mov ecx, edi
not ecx
shr rsi, cl
mov eax, esi
and eax, 1
```

`(block >> (i%64)) & 1ULL;`

- ▶ Fülle Bit-Vector von rechts nach links



- ▶ Assembler-Code:

```
mov ecx, edi
shr rsi, cl
mov eax, esi
and eax, 1
```

Experimente: Bit-Vectoren evaluieren

Live Demo

Aktueller Stand

- ▶ Wir haben einen platzeffizienten Bit-Vektor
- ▶ Wir können Bits schnell lesen

Wie schreiben wir Bits?

- ▶ Viele Bit-Manipulationen und Tricks
- ▶ Bei Interesse:
<https://graphics.stanford.edu/~seander/bithacks.html>

Und jetzt?

- ▶ Rank-Anfragen
- ▶ Finde heraus wieviele Elemente vor Position i (nicht) markiert sind

Design und Analyse: Naive Rank-Datenstrukturen

Zählen bei Anfrage

- ▶ Scannen Bit-Vektor bei jeder Anfrage
- ▶ Dauert länger für große i als für kleine

Zählen bei Konstruktion

- ▶ Speicher den Wert für jedes i
- ▶ Benötigt viel Platz

Live Demo

Design und Analyse: Schnelle und kleine Rank-Datenstruktur

Überlegungen

- ▶ Alles speichern braucht viel Platz
- ▶ Nichts speichern hat lange Anfragen zur Folge
- ▶ **Lösung:** Speicher manche Ergebnisse
- ▶ Aber welche?

Population Count (Popcount)

- ▶ Gibt die Anzahl der 1-Bits für `uint64_t` (und andere) zurück
- ▶ Viele Prozessoren unterstützen Popcount

Entwicklung und Experimente: Popcount

Live Demo

Die Popcount Rank-Datenstruktur

An der Tafel

Basiert auf

Dong Zhou, David G. Andersen, Michael Kaminsky: *Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences*. SEA 2013: 151-163

Design und Analyse: Welche Ergebnisse speichern?

Ergebnis für jedes 512-te Bit (L2)

- ▶ Ergebnisse können beliebig groß sein (`uint64_t`)
- ▶ Benötigt ca. 12,5 % extra Platz

Ergebnis für jedes 2^{32} -te (L0) und 512-te Bit (L2)

- ▶ Sehr wenige Ergebnisse in L0
- ▶ Ergebnisse in L2 können maximal 2^{32} groß sein (`uint32_t`)
- ▶ Benötigt ca. 6,25 % extra Platz

Ergebnis für jedes 2^{32} -te (L0), 2048-te (L1) und fast jedes 512-te Bit (L2)

- ▶ L1 so groß wie L2 zuvor, L2 jetzt 10 Bits (2 mal `uint32_t`)
- ▶ Benötigt ca. 3,125 % extra Platz

Zusammenfassung

- ▶ Effizienter Bit-Vektor
- ▶ Idee für Rank-Datenstruktur
- ▶ Arbeitsweise beim Algorithm Engineering

ABER

*„The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; **premature optimization is the root of all evil** (or at least most of it) in programming.“*

—Donald E. Knuth

Vielen Dank für die Aufmerksamkeit