Ramasort

Marc Goritschnig, Christian Lutnik, Alexander Woda

Algorithmische Optimierung mittels MinHeap

Funktionen: Insert, DeleteMin, ReplaceMin

Laufzeit aller Funktionen in log(heapsize) wobei ‘heapsize’ immer der aktuellen Füllmenge entspricht

Größe: cbrt(n)

Start nodes mit (0, i, sum) $0 < i <= \text{cbrt}(n)$

Optimierung 1 (Heap-Insert Methode) (Grundlage)

Division durch 2 zwischengespeichert

Division durch 2 durch Bitshift ersetzt

```
int Insert(struct entry element)
{
  heapSize++;
  heap[heapSize] = element; /*Insert in the last place*/
  /*Adjust its position*/
  int now = heapSize;
  int now_half = now >> 1;
  while (heap[now_half].value > element.value)
  {
    heap[now] = heap[now_half];
    now = now_half;
    now_half = now_half >> 1;
  }
  heap[now] = element;
  return now;
}
```

Optimierung 2 (Aufbauend auf 1)

-O3 Compiler Flag gesetzt

Optimierung 3 (Aufbauend auf 1)

Suche 2 Ebenen in die Tiefe, ob es genau zwei Elemente mit gleicher Summe gibt

Optimierung der CheckForOneSameSum Funktion durch early return

```
int CheckForOneSameSum(int i)
{
  int c = 0;

  long val = heap[1].value;

  if (2 < heapSize)
  {
    if (heap[2].value == val)
      c++;
  }
  if (4 < heapSize)
  {
    if (heap[4].value == val)
      if (c == 1)
        return 0;
      else
        c++;
  }
  if (5 < heapSize)
  {
    if (heap[5].value == val)
      if (c == 1)
        return 0;
      else
        c++;
  }
  if (3 < heapSize)
```

Optimierung 4 (Aufbauend auf 1)

Optimierung der CheckForOneSameSum Funktion durch

verschachteln der if-Abfragen, sodass keine redundanten Abfragen erfolgen

```
int CheckForOneSameSum(int i)
{
  int c = 0;

  long val = heap[1].value;

  if (2 < heapSize)
  {
    if (heap[2].value == val)
      c++;
    if (3 < heapSize)
    {
      if (heap[3].value == val)
        c++;
      if (4 < heapSize)
      {
        if (heap[4].value == val)
          c++;

        if (5 < heapSize)
        {
          if (heap[5].value == val)
            c++;

          if (6 < heapSize)
          {
            if (heap[6].value == val)
              c++;

            if (7 < heapSize)
            {
              if (heap[7].value == val)
                c++;
            }
          }
```

Optimierung 5 (Aufbauend auf 1)

Optimierung der CheckForOneSameSum Funktion durch eine extra if-Abfrage um zu überprüfen, ob weitere if-Abfragen notwendig sind

```
int CheckForOneSameSum(int i)
{
  int c = 0;

  long val = heap[1].value;

  if (7 < heapSize)
  {
    if (heap[2].value == val)
      c++;

    if (heap[4].value == val)
      c++;

    if (heap[5].value == val)
      c++;

    if (heap[3].value == val)
      c++;

    if (heap[6].value == val)
      c++;

    if (heap[7].value == val)
      c++;
  }
  else
  {
```

Optimierung 6 (Aufbauend auf 1)

```
heap[now] = lastElement;
heap[heapSize + 1].value = -1;
return minElement;
```

Optimierung von Delete: Elemente im heap werden nicht rausgelöscht, sondern auf -1 gesetzt

If-Abfragen in CheckForOneSameSum sind dadurch obsolet

```
int CheckForOneSameSum(int i)
{
  int c = 0;

  long val = heap[1].value;

  if (heap[2].value == val)
    c++;

  if (heap[4].value == val)
    c++;

  if (heap[5].value == val)
    c++;

  if (heap[3].value == val)
    c++;

  if (heap[6].value == val)
```

Optimierung 7 (Aufbauend auf 1)

Optimierung von Delete indem die gelöschten Elemente gleich durch das neue Minimum ersetzt werden

```
struct entry ReplaceMin(struct entry newEntry)
```

Optimierung 8 (Aufbauend auf 7)

Berechnung der Kubikzahlen optimieren, indem diese vor dem if-Statement zwischengespeichert werden

```
// Insert new node if i < j
min.k++;
long sum = cube(min.k) + cube(min.l);
if (min.k < min.l && sum <= n)
{
  min.value = sum;
  ReplaceMin(min);
}
else
{
  DeleteMin();
}
```

Optimierung 9 (Aufbauend auf 7)

Berechnung der Kubikzahlen optimieren, indem diese vorberechnet und in einem Array gecacht werden

```
for (int j = 1; j <= maxNumbers; j++)
{
  long cubeRes = cube(j);
  cubes[j] = cubeRes;
  Insert((struct entry){0, j, cubeRes});
}
```

```
// Insert new node if i < j
min.k++;
long sum = cubes[min.k] + cubes[min.l];
if (min.k < min.l && sum <= n)
{
  min.value = sum;
  ReplaceMin(min);
}
else
{
  DeleteMin();
}
```

Optimierung 10 (Aufbauend auf 7)

Optimierung von DeleteMin durch zwischenspeichern der now * 2 Berechnung, Multiplikation durch Bitshift ersetzen

```
struct entry minElement, lastElement;
int now, child, now2;
minElement = heap[1];
lastElement = heap[heapSize--];
/* now refers to the index at which we are now */
for (now = 1, now2 = 2; now2 <= heapSize; now = child, now2 = now << 1)
{
  /* child is the index of the element which is minimum among both the ch
  /* Indexes of children are i*2 and i*2 + 1*/
  child = now2;
```

Optimierung 11 (Aufbauend auf 10)

Alle int variablen mit long variablen ersetzen

```
long now, child, now2;
```

Optimierung 12 (Aufbauend auf 10)

Minimum Element des Heaps zwischenspeichern

```
struct entry min = heap[1];
// printf("size %ld\n", heapSize);
// printHeap();
if (CheckForOneSameSum(1))
{
  count++;
  // printf("found:       k %ld, l %ld, value %ld, si
  checksum += min.value;
}

// printf("k%ld, l%ld, value %ld\n",min.k, min.l, min

// Insert new node if i < j
min.k++;
if (min.k < min.l && cube(min.k) + cube(min.l) <= n)
{
  min.value = cube(min.k) + cube(min.l);
  ReplaceMin(min);
}
else
{
  DeleteMin();
}
```

Optimierung 13 (Aufbauend auf 12)

Optimierung der Wurzelberechnung

Basierend auf “Hacker’s Delight - Second Edition” by Henry S. Warren, Jr.

```
size_t size_heap(long n)
/* compute a table size that is large enough to keep all I^3+J^3<n */
{
  int s;
  unsigned int y;
  unsigned long b;

  y = 0;
  for (s = 63; s >= 0; s -= 3)
  {
    y += y;
    b = 3 * y * ((unsigned long)y + 1) + 1;
    if ((n >> s) >= b)
    {
      n -= b << s;
      y++;
    }
  }
  return y;
}
```

Optimierung 14 (Aufbauend auf 13)

Explizites Function Inlining für kleiner Funktionen

```
long inline cube(long n)
```

```
int inline CheckForOneSameSum(int i)
```

Optimierung 15 (Aufbauend auf 14)

ALLE Variablen global machen

Keine gute Idee

```
// size_heap
int s;
unsigned int y;
unsigned long b;

// HEAP    ////////////////////////////////////////////////////

// Init
int heapSize;
struct entry *heap;

// Insert
int now;
int now_half;

// CheckForOneSameSum
int c = 0;
long val;

// DeleteMin
struct entry minElement_d, lastElement_d;
int now_d, child_d, now2_d;
```

Optimierung 16 - 18 (Aufbauend auf 14)

Compiler Flags anpassen

```
CFLAGS = -Ofast -DDONT_USE_VOL -Wall
```

Mit -Ofast versucht der Compiler das Executable so schnell wie möglich zu machen, auch wenn dabei manche Standards verletzt werden

Mit -DDONT_USE_VOL versucht der Compiler volatile Variablen zu vermeiden, was in manchen Fällen mehr Optimierungen zulässt.

Optimierung 19 (Aufbauend auf 18)

Summenberechnung zwischenspeichern

```
// Insert new node if i < j
min.k++;
sum = cube(min.k) + cube(min.l);

if (min.k < min.l && sum <= n)
{
  min.value = sum;
  ReplaceMin(min);
}
```

Potentielle Optimierung 1

Modulo Werte der Summen ausnützen

Ergebnisse bis 999999259966294: https://www.numbersaplenty.com/set/taxicab_number/more.php#moduli

DOES NOT WORK - PROPERTY OF CUBES, NOT RH-NUMBERS!

n\r	0	1									
2	558502	312348	2								
3	489438	196536	184876	3							
4	532732	156521	25770	155827	4						
5	208643	165249	165548	165897	165513	5					
6	312567	79511	128910	176871	117025	55966	6				
7	552468	42862	116357	0	0	115974	43189	7			
8	522613	78786	12877	77832	10119	77735	12893	77995	8		
9	489438	166235	27944	0	0	0	0	30301	156932	9	
10	131671	58616	106422	58599	106478	76972	106633	59126	107298	59035	10
11	78767	79166	79200	79274	79333	79187	79043	79172	79247	79158	79303

Potentielle Optimierung 1

Kubik Berechnung eliminieren durch Differenzberechnungen

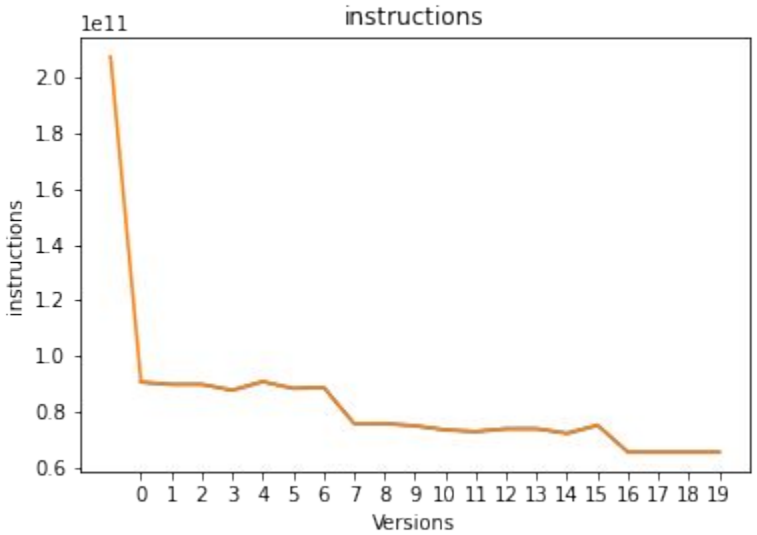
Siehe https://www.quora.com/Is-there-a-pattern-between-cubed-numbers

$$(A+1)^3 = A^3 - (A-1)^3 + A \cdot 6$$

$$DIF = DIF + A \cdot 6$$

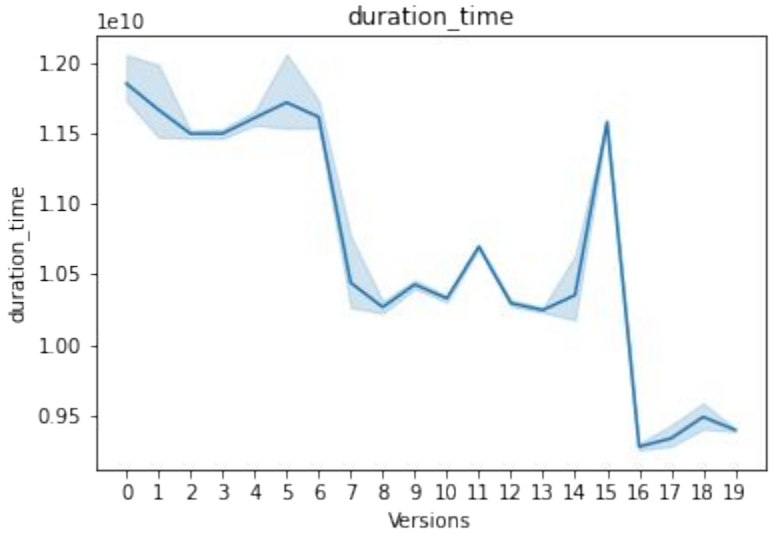
Plots

Ohne Baseline

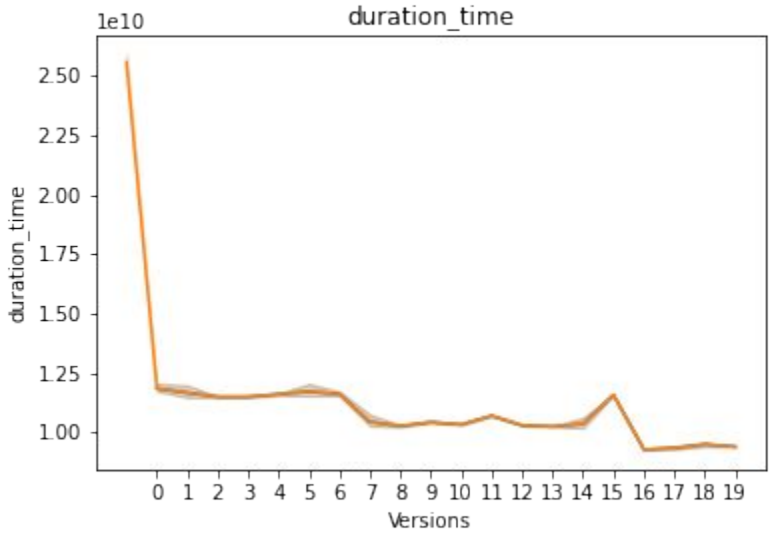


Mit Baseline

Plots



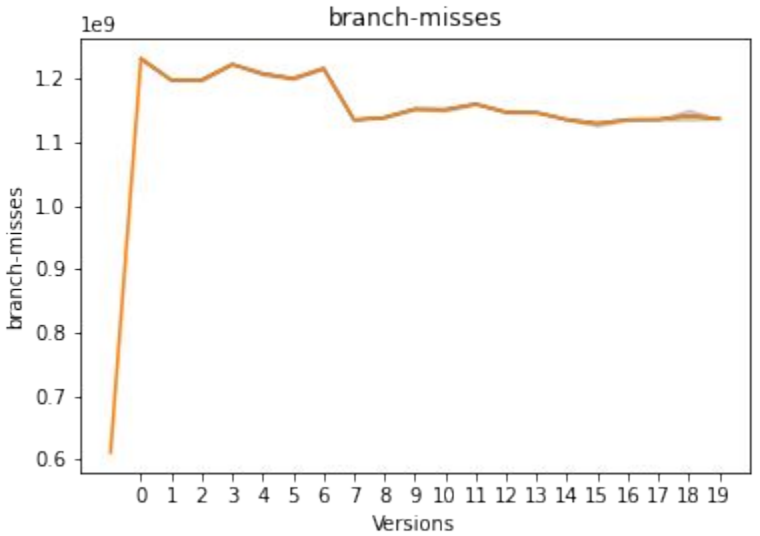
Ohne Baseline



Mit Baseline

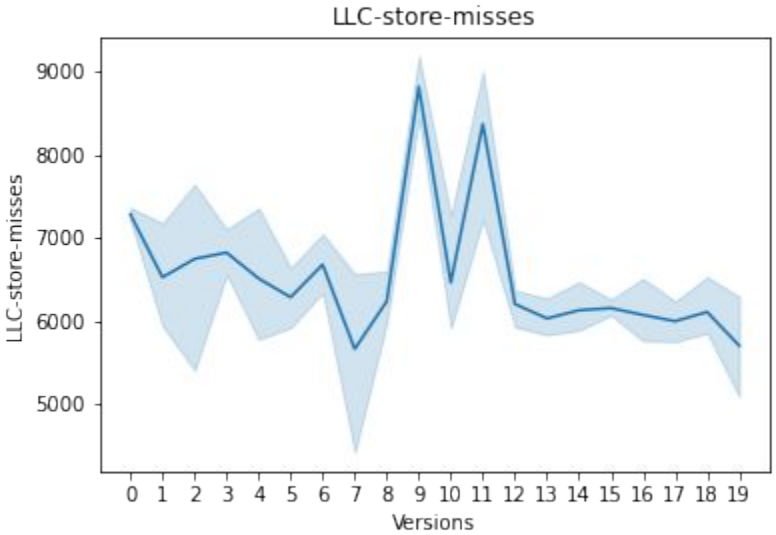
Plots

Ohne Baseline

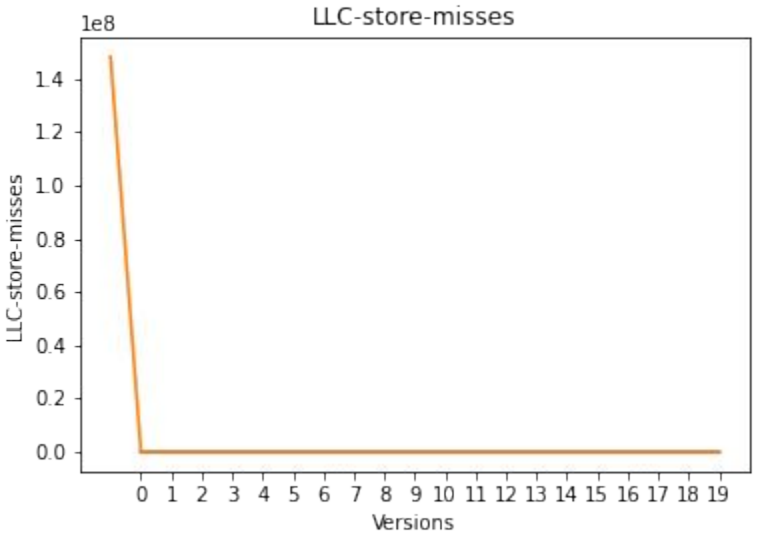


Mit Baseline

Plots



Ohne Baseline



Mit Baseline

Summary

- Viel Trial-And-Error
- Variable Laufzeit und auf der g0

- Runtime
  - 25,57 s —> 9,28 s
  - Faktor 2,75

- Memory Allocations (Valgrind Massif Tool)
  - 3 713 272 656 byte —> 344 704 byte
  - Faktor 10 772