Nov 7th, 5:15 PM - 5:30 PM

Efficient Algorithm for solving 3SUM problem

Muhamed Retkoceri
University for Business and Technology, muhamed.retkoceri@gmail.com

Efficient Algorithm for solving 3SUM problem

Muhamed Retkoceri
Faculty of Computer Science and Engineering
University for Business and Technology
muhamed.retkoceri@gmail.com

Abstract. In this paper is presented an algorithm for solving 3SUM problem efficiently in general computation model. The algorithm is based on sorting and splits the task into sub-tasks where this approach enables the algorithm to run concurrently at the high-level of computing. The algorithm is O(n2) and running sequentially achieves at least ~1/5 n2 number of basic necessary accesses of data structures. In this paper is also presented a comparison of running performances in practice between the new algorithm and the current most famous algorithm for 3SUM which is in-place and also based on sorting.

Keywords: 3SUM, Concurrent 3SUM, Algorithm, Concurrent Approach

1 Introduction

In computational complexity theory, the 3SUM problem asks if a given set of n integers, each with absolute value bounded by some polynomial in n, contains three elements that sum to zero. [1,2,3].
The generalized version, rSUM, asks the same question of r elements. [1,2,3]. A naive solution would be to search for all triples with three indexes where time complexity would be O(n3). Better algorithm should achieve it in O(n2).There is a sub-quadratic algorithm for special models of computation. [5].
"There are a algorithms based on sorting with partial information". [1]An algorithm for solving 3SUM is presented based on the Fast Fourier Transformation taking into assumption that absolute values of n numbers are smaller than n2/log n. [1,6].Solutions for general version of the problem are found in [1][2].
"The 3SUM problem was initially set in [2]. Gajentaan and Overmars collecteda large list of geometric problems, which may be solved in an order of quadraticcomplexity, and nobody knows, how to do it faster [2]."[1].3SUM is an important problem as it still remains to be an open problem in theoretical computer science. 3SUM is found all over computational geometry.[2].The algorithm presented on this paper is still quadratic but it is at least three times faster than the best known quadratic algorithm which is presented in [7]. Running concurrently the algorithm would be additionally three to four times faster. In terms of memory complexity the algorithm is O(n). It consumes additional linear memory but improves running performance by a factor.

2 The algorithm

Suppose the input array is S[0 … n-1] of size n. First S is sorted and then is separated into arrays of negatives and positives.
Let p be the number of positives and the array of negatives N[0 … n-p] in descending order and also the array of positives P[0 … p] in ascending order.
Additionally let HTP(k,v) be the HashTable of Positives and HTN(k,v) the HashTable of Negatives, where k is the key and v is the value.
All elements of P are put as keys in HTP in ascending order and it's indexes as values, also all elements of N are put as keys in HTN in descending order and it's indexes as values in ascending order, so i = HTN.get(N[i]) .

The algorithm first checks for each negative number to every positive if there exists the third positive number to sum up to zero and vice versa.

```
for i:=0 to n-p {
  a := N[i]
  for j:=0 to p {
    b:= P[j]
    c:= -(a+b)
    if c<b then break inner loop
    if HTP contains key(c) {
      if b=c and j = HTP.get(c) then break inner loop
      else save triple a,b,c
    }
  }
}
for i:0 to p {
  a := P[i]
  for j:=0 to n-p {
    b := N[j]
    c := -(a+b)
    if c>b then break inner loop
    if HTN contains key(c) {
      if b=c and j = HTN.get(c) then break inner loop
      else save triple a,b,c
    }
  }
}
```

The statement if b=c and j = HTP.get(c) enables the algorithm to distinguish duplicate elements so all elements are considered as unique.
If the input is distinct simply that if statement can be removed and inside the nested loop the pruning if statements must be modified to if c<=b and if c<=b.

2.1 Concurrent approach

The concurrent approach to solve the task requires separation of the input into odd and even numbers besides into negatives and positives.

Suppose indexes 1,2,3,4 represent simultaneous processes, so the algorithm would be:

Assume the input is the set $S \in \mathbb{Z}$ and a, b, c, d ∈ S where $a \in \{2n < 0 : n \in \mathbb{Z}\}$,
$b \in \{2n+1 < 0 : n \in \mathbb{Z}\}$, $c \in \{2p \geq 0 : p \in \mathbb{Z}\}$ and $d \in \{2p+1 \geq 0 : p \in \mathbb{Z}\}$.

1. ∀(b,d) check if ∃ -(b+d) and ∀(b,c) check if ∃ -(b+c)
2. ∀(a,c) check if ∃ -(a+c) and ∀(a,d) check if ∃ -(a+d)
3. ∀(d,a) check if ∃ -(d+a) and ∀(d,b) check if ∃ -(d+b)
4. ∀(c,b) check if ∃ -(c+b) and ∀(c,a) check if ∃ -(c+a)

The above algorithm contains eight loops which four of them would run at the same time. The first loop: ∀(b,d) check if ∃ -(b+d), checks for every odd negative to every odd positive if there exists an even positive third pair to sum up to zero. The algorithm is searching on different groups of elements at the same time possibly speeding the algorithm up to 400%.

3 The comparison

The famous algorithm is as follows: [7]. Let's assume input array is S[0 … n-1].

```
Sort(S)
for i:=0 to n-3 {
  a := S[i]
```

```
 k := i+1
 l := l-1
 while(k<l) {
  b := S[k]
  c := S[l]
  if a+b+c = 0 {
   save triple a,b,c
   k := k+1
   l := l-1
  }
  else if a+b+c > 0 then l := l-1
  else k := k+1
 }
}
```

Fig. 3. In this picture is compared the execution time in general computers with different random input for both algorithms. Algo_1 is the presented algorithm and clearly beats the in-place algorithm. The average running time is taken from different machines and the graph represents the size of input n in proportion to CPU time in nanoseconds.

4 Conclusions

Using distinct input and HashSet lookups the algorithm is supposed to run even faster.
Further improvements can be done if somehow is possible to know if in small ranges between b and c there is no other pair to sum up to 3 so those indexes could be easily skipped. When c is found on HashMap, the associated value which is index can be checked if between b and c is only one element where it can be safely skipped. Using nested HashMaps and Trees maybe it might be possible to skip few more indexes.

References

1. Valerii Sopin, “A new algorithm for solving the rSUM problem”, arXiv:1407.4640v4.
2. A. Gajentaan and M. Overmars, “On a class of O(n2) problems in computational geometry”, Computational Geometry: Theory and Applications, 1995, № 5, 165–185.
3. Erickson, Jeff, “Lower bounds for linear satisfiability problems”, Chicago Journal of Theoretical Computer Science, 8, 1999.
4. King, James, “A Survey of 3sum-Hard Problems”, www.cs.mcgill.ca, king@cs.ubc.ca, 2004.
5. I. Baran, E. Demaine and M. Patrascu, “Subquadratic algorithms for 3SUM”, Lecture Notes in Computer Science, 2005, № 3608, 409—421.
6. T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to Algorithms, MIT Press and McGraw-Hill, 2001.
7. Michael Hoffmann “12. Visibility Graphs and 3-Sum”, Lecture on Monday 9th November, Hoffmann@inf.ethz.ch, 2009.