International Journal of Computer & Information Technologies (IJOCIT)
www.ijocit.org & www.ijocit.ir
ISSN= 2345-3877

A Note on the 3-Sum Problem

Keivan Borna[1], Zahra Jalalian[2]
Faculty of Mathematical Sciences and Computer[1], Faculty of Engineering[2]
Kharazmi University[1& 2]

borna@khu.ac.ir [1] , jalalian@khu.ac.ir[2]

Abstract: The 3-Sum problem for a given set S of integers is subject to find all three-tuples ***(a, b, c)*** for which ***a + b + c = 0***. In computational geometry many other problems like motion planning relate to this problem. The complexity of existing algorithms for solving 3-Sum are ***O(n²)*** or a quotient of it. The aim of this paper is to provide a linear hash function and present a fast algorithm that finds all suitable three-tuples in one iteration of ***S***. We also improve the performance of our algorithm by using index tables and dividing ***S*** into two negative and non-negative parts.

Keywords: 3-Sum Problem, Computational Complexity, Linear Hash Function, Motion Planning.

1. Introduction

The 3-Sum problem for a given set ***S*** of ***n*** integers asks whether there exist a three-tuples of elements from ***S*** that sum up to zero. A problem ***P*** is 3-Sum-hard if every instance of 3-Sum of size ***n*** can be solved using a constant ***O(n²)*** additional time. One can think of a 3-SUM-hard problem in many interesting situations including incidence problems, separator problems, covering problems and motion planning. Obviously by testing all three-tuples this problem can be solved in ***O(n³)*** time.

Furthermore if the elements of S are sorted then we can use Algorithm 1 with $O(n^2)$ complexity.

It is interesting to mention that Algorithm 1 is essentially the best algorithm known for 3-Sum and it is believed that the problem cannot be solved in sub-quadratic time, but so far this has been proven in some very restricted models of computation only, such as the linear decision tree model. In fact Erickson [4, 5] proved an $\Omega(n^2)$ lower bound in the restricted linear decision tree model. This model is based on the transpose of the transformation presented in [3] that maps each point ***(a, b)*** to the line ***y = ax + b*** and vice-versa. However, the problem remained unsolved in general for other computational models.

In [1] the authors presented a sub-quadratic algorithms for 3-Sum. More precisely on a standard word RAM with ***w-bit*** words, they obtained a running time of

T $O(n^2/\max\{lg^2n/(lglgn)^2, \omega/lg^2\omega\} + sort(n))$.

map h that was already introduced in [2]. In fact for a random odd integer a on w bits, the hash

```
Data: A sorted set S of integers
Result: A three-tuple (a, b, c) for whicl
for i = 1, ··· , n − 2 do
    j=i, k=n-1;
    while k > j do
        if s_i + s_j + s_k = 0 then
            print s_i, s_j, s_k;
            j=j+1;
            k=n-1;
        end
        if s_i + s_j + s_k > 0 then
            k=k-1;
        else
            j=j+1;
        end
    end
end
```

Algorithm 1: An $O(n^2)$ algorith

function *h* maps each ***x*** to the first ***s = lgw*** bits of ***a *x***. In the second section of [1] for a given σ, the authors found suitable a, b for which σ = ***a+b***. In fact they proved that if σ = ***a+b*** then ***h(σ) = h(a) ⊕ h(b) ⊕ {0,1}*** and if $\sigma \neq$ ***a+b*** then ***h(a) h(b) ⊕ h(- σ) ϵ {0, -1,-2 }*** is only true with small probability, i.e., ***if h(a) ⊕ h(b) ⊕ h(-σ) ∉ {0, -1, -2}*** then there is no ***(a, b)*** for which ***σ = a+b***. Notice that the operator ⊕ is modulo 2^s and we have ***h(a) ⊕ h(b ⊕ h(c) ϵ h(a+b+c) ⊕ {0, 1, 2}.***

Furthermore since multiplication is linear, removing the first ***w-s*** bits of the multiplication makes h to be non-linear. The reason to use the factor ***{0, 1, 2}*** is to make this map linear. We refer the interested reader to see [2, 6] for more information about this hash function which is known as Universal hashing.

In this paper we apply a linear hash function h that uses only one substraction operation. More precisely for a sorted array ***S*** of length ***n*** we define ***h*** via ***h(i) = S[i] - S[1]***. Now we construct a new array ***R*** of length ***S[n-2] - S[1] +1*** for which the relation ***R[h(i)] = S[i]*** between the indices and values of its elements is established. In fact ***R*** indicates a set that is created by h and knowing the value ***h(i)*** one can obtain ***S[i]*** as ***S[i] = h(i) + S[1].*** Now applying our algorithm and by only one iteration over ***S*** one can find all three-tuples ***(a, b, c)*** for which ***a + b + c = 0***.

The organization of this paper is as follows. In Section 2 our proposed algorithm and its complexity analysis are presented. In Section 3 the performance of our algorithm by using index tables instead of arrays and dividing ***S*** into two specific parts are improved. Finally Section 4 is devoted to some conclusions and future works.

2. Our Algorithm

In this section we present our proposed algorithm. For the ease of reader more details about this algorithm will be given during an example. Let ***S*** be a sorted array of integers of length n. We first define a hash function ***h*** with ***h(i) = S[i]- S[1].*** Then we construct a sorted array ***R*** of length size ***R = S[n-2]- S[1]+1*** initialized with ***S[0] - 1***. Then we allocate members of ***S*** in ***R*** via ***h*** and the formula ***R[h(i)] = S[i]***. Now let ***indexa = 0, indexc = n - 1; a = S[indexa], c = S[indexc].***

Repeat the following commands while ***indexa < n - 3:***

1. Let ***indexb = - (a + c) - S[1].*** In fact index ***b*** represents the index of ***b*** in array ***R*** for which ***a + b + c = 0***.
2. If ***indexb ≥ 0, indexb < sizeR*** and ***R[indexb]≠ S[0]-1*** then ***let b =***

R[indexb]. Now if ***a+b+c = 0*** and ***b > a*** and ***b < c*** then we have found a suitable three-tuples. In order to find the other solutions let ***indexc = indexc - 1*** and do the next repetition of the loop.

3. If ***(a + c < S[0]*** and ***a + c > S[n - 1])*** or ***a ≥ c***, then there is no suitable value for b for which ***a + b + c = 0*** and so a must be changed with larger values in ***S***. Thus in order to find a solution let ***indexa = indexa + 1, a = S[indexa]; indexc = n - 1*** and ***c = S[indexc]*** and do the next repetition of the loop.

4. Else if ***a < c***, then ***c*** is too large and so ***c*** must be changed with smaller values in

S. That is let ***indexc = n - 1*** and ***c = S[indexc].***

In the following, Algorithm 2 computes all suitable three-tuples for the ***3-Sum*** problem:

As an example if **S** is an array with 8 elements -25, -10, -7, -3, 2, 4, 8, 10, then ***R*** has 19 elements and index of each element of it will be computed via ***S[i] - S[1].*** Thus ***R[0] = -10 , R[3] = -7, … , R[18] = 10*** and the other cells will be filled with ***S[0] - 1 = -26***. Hence ***R = -10, -26, -26, -26, -7, -26, … , -26,*** 8. Now let a (and ***c***) be the first (and last) element of S. That is, ***a = S[0] = -25, c = S[n-1] = 10. Let j = (-(a + c) - S[1]) = (-(-25 + 10) - (-10)) = 25*** and since ***j*** is not in the range of indices of ***R***, for this a we cannot find **b, c.**

But since ***a + c = -15 < 0*** hence ***i = i + 1 = 1***, ***a = S[i] = -10***. Then the new value for ***j*** is ***j = (-(a+c) - S[1]) = (-(-10+10)-(-10)) = 10***. Since ***R[10] = -26***, no value for b for which ***a+b+c = 0*** is found. On the other hand since ***a+c = 0*** so we should seek for a smaller value of c. Thus let ***l = l - 1 = 6, c = S[l] = 8***. Since ***j = (-(a + c) - S[1]) = (-(-10 + 8) - (-10)) = 12, b = R[12] = 2*** and ***a+b+c = -10+2+8 = 0*** thus we have found a suitable three-tuples. Our algorithm reports the other solution as ***-7 + -3 + 10 = 0.***

```
Data: A sorted set S of n integers
Result: All (a, b, c) for which a + b + c = 0.
indexa = 0, indexc = n-1;
a = S[indexa], c = S[indexc];
sizeR = S[n-2] - S[1] + 1;
while indexa < n - 3 do
    indexb = -(a+c)-S[1];
    if 0 ≤ indexb < sizeR and R[indexb] ≠ S[0]-1 then
        b = R[indexb];
        if a + b + c = 0 and c > b > a then
            print a, b, c;
        end
        indexc = indexc - 1;
        c = S[indexc];
    end
    if S[n − 1] < a + c < S[0] or a ≥ c then
        indexa = indexa + 1;
        a = S[indexa];
        indexc = n-1;
        c = S[indexc];
    end
    else if a < c then
        indexc = indexc - 1;
        c = S[indexc];
    end
end
```

Algorithm 2: Our algorithm for finding all solutions for the 3-Sum problem

One can see that this algorithm finds all the three tuples ***a, b, c*** for which ***a+b+c = 0*** in one iteration of the given array. Furthermore in this example one can note that if in each loop three conditional statements are going to run (in the middle case), then we obtain 24 statements totally. Whereas using Algorithm 1 this amount will be 35. One of the advantages of our algorithm is that the collisions in our hash function is impossible, this is because elements of set ***S*** and thus ***R*** will not repeat.

3. Improving the performance of our Algorithm

Two possible limitations of our algorithm are the use of extra memory and the number of comparisons. In this section we provide solutions to overcome these two limitations.

3.1. Using data file plus index table instead of array

In this subsection we improve the performance of our algorithm when the number of elements of ***S*** is very large using a bitmap. In fact when the size of array ***S*** is large and it would not fit in the main memory, we can use a file located in the auxiliary memory and an index table which is in the main memory. The data in the file put in blocks and index table shows the largest amount of data in each block. In this way we can quickly access to the address of data and we can check if there is any ***b**" for ***a+ c*** such that ***a+b+ c =0***.

The bit array ***R*** is a word in RAM consisting of ***m := S[n-2]-S[1]+1*** bits. We initialize all bits of ***R*** with zero. In order to map elements of ***S*** in ***R***, we consider ***R*** as a word with at least m bits in RAM. Then each bit indicates a number in ***S***. If this bit is one (zero), then we deduce that the number exists (does not exist) in ***S***. Then for members of S we use the following formulas:

$$\alpha = -S[0], h(S[i]) = S[i] + \alpha, R[h(S(i)] = 1$$

As an example let ***S = {-25, -10, -7, -3, 2, 4, 8, 10}***. If ***a = -10, c = 8***, then b should satisfy ***b = -(-10 + 8) = 2***. Therefore using the bitmap we refer to the bit number ***b+α= 2+10=12*** and if this bit is one we conclude that b exists in S and henceforth we have found a suitable three-tuples in ***S***.

3.2 Dividing S into two parts

When the number of elements of ***R*** is very large and we cannot store S in the main memory, another useful approach can be applied. As a matter of fact we can put the array R into a file and use two index tables for choosing the values of ***a*** and ***c***. Note that in order to have ***a + b + c =***

0, at least one of ***a, b, c*** should be negative. Thus we can divide S into two subsets S_1 and S_2 for choosing negative and non-negative values and store them in the main memory. The following algorithm computes all suitable three-tuples for the 3-Sum problem very quickly. Let midS denote the index of the first non-negative element of ***S***. Let $\alpha \epsilon 2\ S_1; c \epsilon\ S_2$, then using the relation ***indexb = - (a+c)+S[1]*** we can decide whether b exists in the file or not. If ***-(a + c) < S[1]*** then we have to move a to the next element of S_1, or if ***-(a + c) > S[n - 1]*** then there is no suitable b for the current values of ***a, c*** and we have to take another elements from S_1 and S_2. As an example if ***S = {-25, -10, -7, -3, 2, 4, 8, 10},*** then the two index tables are ***S1 = {-25, -10, -7, -3}***, S_2 ***= {2, 4, 8, 10}***. Our algorithm proceeds and reports the following two solutions: - 10 + 2 + 8 = 0 = -7 + - 3 + 10. In the following we present the improved version of our algorithm for finding all solutions for the ***3-Sum*** problem. Note that since the sets for choosing the values for ***a*** and ***c*** are being smaller, the amount of comparisons and thus the running time of our algorithm will decrease.

We conclude our discussions in Sections 2 and 3 in the following theorem. Theorem 1: Algorithm 3 computes all solutions for the ***3-Sum*** problem faster than Algorithms 1 and 2.

Finally we compare our Algorithms 2 and 3 with Algorithm 1. We generated 100 sets all of size 20 of random integers in the range [-50, 50] and counted the amount of operations that each algorithm are doing. In figures 1 and 2 (the outputs of a Java program) the leftmost, the middle and the rightmost histograms count the amount of operations that each of Algorithm 1, 2 and 3 are doing, histograms count the amount of operations that each of Algorithm 1; 2 and 3 are doing.

In Figure 2 we draw the histograms for the average amount of operations that each algorithm are doing.

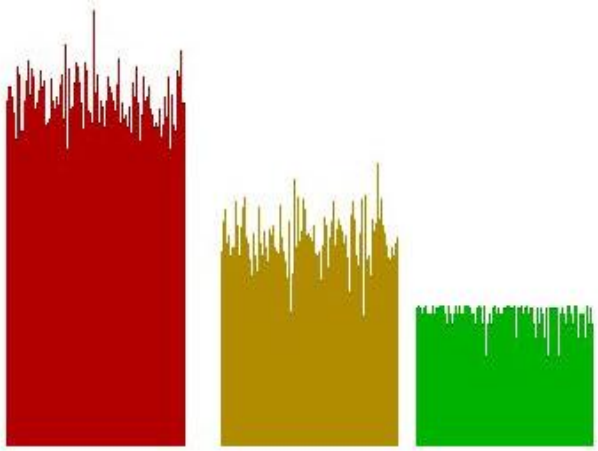

Figure 1: The histogram of amount of operations that Algorithms 1, 2 and 3 are doing

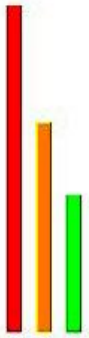

Figure 2: The histogram of avarage amount of operations that Algorithms 1, 2 and 3 are doing in 100 tests.

4. Discussion and Future Works

In this paper a fast and optimized algorithm for finding all solutions for the 3-Sum problem is presented. Further works for generalizing this algorithm to rational and complex numbers are under progress.

```
Data: A sorted set S of n integers
Result: All (a, b, c) for which a + b + c = 0.
indexa = 0, indexc = n-1;
a = S[indexa], c = S[indexc];
sizeR = S[n-2]-S[1]+1;
while indexa < midS do
    indexb = -(a+c)-S[1];
    if 0 ≤ indexb < sizeR and R[indexb] ≠ S[0]-1 then
        b = R[indexb];
        if a + b + c = 0 and c > b > a then
            print a, b, c;
        end
        indexc = indexc - 1;
        if indexc < midS then
            indexa = indexa + 1;
            a = S[indexa];
            indexc = n-1;
        end
        c = S[indexc];
        continue;
    end
    if S[n − 1] < a + c < S[0] or a ≥ c then
        indexa = indexa + 1;
        a = S[indexa];
        indexc = n-1;
        c = S[indexc];
    end
    else if a < c then
        indexc = indexc - 1;
        if indexc < midS then
            indexa = indexa + 1;
            a = S[indexa];
            indexc = n-1;
        end
        c = S[indexc];
    end
end
```

Algorithm 3: Our improved algorithm for finding all solutions for the 3-Sum problem

References

[1] I. Baran, E. D. Demaine, and M. Patrascu, Subquadratic algorithm for 3-SUM: Proc. 9th Worksh. Algorithms & Data Structures, Springer, Berlin/Heidelberg 3668/2005 (2005), 409 - 421.
[2] M. Dietzfelbinger, Universal hashing and k-wise independent random variables via integer arithmetic without primes: Lecture Notes in Computer Science, Proc. 13th Symposium on Theoretical Aspects of Computer Science (1996), 569- 580.
[3] H. Edlesbrunner, J. ORourke, and R. Seidel, Constructing arrangements of lines and hyperplanes with applications: Lecture Notes in Computer Science, SIAM. J. Computer. 15 (1986), 341-363.
[4] J. Erickson, Lower bounds for fundamental geometric problems, PhD thesis, University of California at Berkeley, 1996.
[5] , Lower Bounds for Linear Satisfiability Problem: Chicago Journal of Theoretical Computer Science 8 (1999).
[6] M. N. Wegman and J. L. Carter, New classes and applications of hash functions, Proc. 20th IEEE FOCS (1979), 175-182.

Authors Profile:

Dr. Keivan Borna is an Assistant Professor at the Department of Computer Science in Faculty of Mathematics and Computer Science of Kharazmi University of Tehran since 2008. He completed his Ph.D. in September 2008 from the Department of Mathematics, Statistics and Computer Science of University of Tehran in Computational Commutative Algebra. He was a visiting scholar in ``Dipartemento di Matematicha, Universita' di Genova- Italia" and ``Department of Mathematik and Informatik at Essen University, Germany'', from Sep. 2007 to Apr. 2008 during his graduate work. He is very interested in studying and researching in interdisciplinary topics like "Cryptography", "Approximate Algorithms", "Musical Data Analysis", "Motion planning" and "Computational Geometry". He published some papers in such areas. He is the author of the ``Advanced Programming in JAVA'' (in Persian) and is the member of ``Elite National Foundation of Iran''.

****

Zahra Jalalian is a faculty

member at the department of Engineering, Kharazmi University, Tehran, Iran. She got her master degree from Concordia University, Montreal, Canada. Zahra has more than a decade of teaching experience on software engineering and computer science undergraduate courses. Also she has some accepted articles in software engineering and computer science conferences. Zahra's research interest is in the area of algorithm design and artificial intelligence.