A 3SUM algorithm on the binary addition level of atomic generated real numbers
-
Discussion doc: checkValid ns values improvement

Carolin Zöbelein*

September 26, 2020

DRAFT

Abstract

In this notes we want to discuss some possible improvements for `ns` value checking in our original paper.

Keywords: 3SUM Problem, Real Numbers, Irrational Numbers, Data Structures, Binary Representation, Data Storage Representation, Nonnumerical Algorithm, Complexity
ACM Subject Classes: E1, E2, F1.3, F2.2

Contents

Preamble

The following content is a sketch for discussion purposes only, without warranty for mathematical completeness.

*The author believes in the importance of the independence of research and is funded by the public community. If you also believe in this values, you can find ways for supporting the author's work here: `https://research.carolin-zoebelein.de/funding.html`, Email: `contact@carolin-zoebelein.de`, PGP: D4A7 35E8 D47F 801F 2CF6 2BA7 927A FD3C DE47 E13B, `https://research.carolin-zoebelein.de` LaTeXsource availabe at `https://github.com/Samdney/paper-3sum`, `id: paper_0003`, 😎

1 Introduction

The final complexity of the given algorithm leads to an $U^{1.58}$ part with $U := 2^b$. The power of 1.58 results from the valid rule $\mathcal{C}$ with a given `ns` combination checks. In this discussion doc, we want to show, how this part of the algorithm can be improved.

2 Background

Considering the possible $\mathcal{C}$ rules at one specific T^{string} node, we have the following possible successor $\mathcal{C}$ pairs at bit position q for zero sum considerations:

1. If we want to get a 0 at bit position q:
   (a) $\mathcal{C}^{0}_{[q],3} := (3,0,0,3)$ and $\mathcal{C}^{1}_{[q],3} := (1,2,1,3)$
   (b) $\mathcal{C}^{10}_{[q],4} := (0,3,10,4)$ and $\mathcal{C}^{1}_{[q],4} := (2,1,1,4)$
   (c) $\mathcal{C}^{10}_{[q],5} := (1,2,10,5)$ and $\mathcal{C}^{1}_{[q],5} := (3,0,1,5)$
2. If we want to get a 1 at bit position q:
   (a) $\mathcal{C}^{0}_{[q],3} := (2,1,0,3)$ and $\mathcal{C}^{1}_{[q],3} := (0,3,1,3)$
   (b) $\mathcal{C}^{1}_{[q],4} := (1,2,1,4)$ and $\mathcal{C}^{0}_{[q],4} := (3,0,0,4)$
   (c) $\mathcal{C}^{1}_{[q],5} := (2,1,1,5)$ and $\mathcal{C}^{10}_{[q],5} := (0,3,10,5)$

We can see that we get the two possible cases:

$$\mathcal{C}^{\alpha}_{[q]} := (3,0) \quad \text{and} \quad \mathcal{C}^{\beta}_{[q]} := (1,2) \tag{1}$$

with worst case checks (see original paper):

1. For $\mathcal{C}^{\alpha}_{[q]}$ $(3,0)$:
   (a) $child_1.count \geq 1 \wedge child_3.count \geq 1 \wedge child_5.count \geq 1$
2. Case $\mathcal{C}^{\beta}_{[q]}$ $(1,2)$:
   (a) $child_1.count \geq 1 \wedge child_4.count \geq 1 \wedge child_6.count \geq 1$
   (b) $child_2.count \geq 1 \wedge child_3.count \geq 1 \wedge child_6.count \geq 1$
   (c) $child_2.count \geq 1 \wedge child_4.count \geq 1 \wedge child_5.count \geq 1$

and

$$\mathcal{C}^{\alpha}_{[q]} := (0,3) \quad \text{and} \quad \mathcal{C}^{\beta}_{[q]} := (2,1) \tag{2}$$

with worst case checks (see original paper):

1. Case $\mathcal{C}^{\alpha}_{[q]}$ $(0,3)$:
   (a) $child_2.count \geq 1 \wedge child_4.count \geq 1 \wedge child_6.count \geq 1$
2. Case $\mathcal{C}^{\beta}_{[q]}$ $(2,1)$:
   (a) $child_1.count \geq 1 \wedge child_3.count \geq 1 \wedge child_6.count \geq 1$
   (b) $child_1.count \geq 1 \wedge child_4.count \geq 1 \wedge child_5.count \geq 1$
   (c) $child_2.count \geq 1 \wedge child_3.count \geq 1 \wedge child_5.count \geq 1$.

3 The improvements

Now, we explain how we can do some improvements regarding the valid combinations checking.

3.1 Check count values in T

In our original paper, we wrote that we have to check each node in T if their *count* values are at least 1, 2 and 3. We want to extend this checks to the following total set of checks for each node in T instead.

1. $node.count == 0$
2. $node.count \geq 1$
3. $node.count \geq 2$
4. $node.count \geq 3$
5. $node.count == node.parent.count$

With this, we set a flag in the following way for each node in T:

1. If $node.count == 0$ is true:
   Set flag $node.flag = 0$.
2. Else if $node.count \geq 1$ is true and $node.count != node.parent.count$ is true:
   Set flag $node.flag = 1$.
3. Else if $node.count \geq 1$ is true and $node.count == node.parent.count$ is true:
   Set flag $node.flag = p$, with p be a fixed prime number.

To do this additional checks and to set the flags don't change the time complexity of T generation with $\mathcal{O}(nb)$.

3.2 Symmetry of child nodes

We can use the symmetry of characteristics of the two child nodes $child_i := node.zero$ and $child_j := node.one$ of a common direct parent node *node*.

We know if the cardinal number of list elements of one child node equals the cardinal number of list elements of its parent, then the list of elements of the other child node has to be empty. With this we can derive the following flag maps between the two child nodes:

$$\begin{array}{rcl} child_i.flag = p & \mapsto & child_j.flag = 0 \\ child_i.flag = 1 & \mapsto & child_j.flag = 1 \\ child_i.flag = 0 & \mapsto & child_j.flag = p. \end{array} \tag{3}$$

3.3 Single `ns` check improvement

Now, we look at our possible checks again. We start with the case $\mathcal{C}^{\alpha}_{[q]} := (3,0)$ and $\mathcal{C}^{\beta}_{[q]} := (1,2)$ and the flags for the necessary checks. We write them in short

1. For $\mathcal{C}^{\alpha}_{[q]}(3,0)$:
   (a) (f_1, f_3, f_5)
2. Case $\mathcal{C}^{\beta}_{[q]}(1,2)$:

(a) (f_1, f_4, f_6)

(b) (f_2, f_3, f_6)

(c) (f_2, f_4, f_5)

We know that the belonging nodes of f_1 and f_2 have the same parent, the nodes of f_3 and f_4 have the same parent and the nodes of f_5 and f_6 have the same parent.

Assume we know (f_1, f_3, f_5). We see that the three other flag triples differ from this one always by two flags which belong to the same parent like the ones of the first triples. For example we have $(f_1, f_3 \mapsto f_4, f_5 \mapsto f_6) = (f_1, f_4, f_6)$. This means, if we know (f_1, f_3, f_5), together with the mapping 3, we immediately also know the other three triples.

Additionally, we can directly derive from a given triple (f_i, f_j, f_k) with $f_i, f_j, f_k \in \{0, 1, p\}$ if the belonging check is true or false, we can derive an unique mapping for each given (f_1, f_3, f_5) setting to a boolean 4-tuple (c_1, c_2, c_3, c_4), $c \in \{\text{true}, \text{false}\}$, telling us which of the four checkings is true and which ones are false.

Since, we have three possible values for each f, we can define one matrix of size $3 \times 3 \times 3$. For a given configuration of (f_1, f_3, f_5), we can directly derive the belonging (c_1, c_2, c_3, c_4). To make it much more efficient we can, instead of saving just the boolean values c, directly write the saving instruction of `ns` as general function depending on the given node pointers, in our 4-tuple.

Finally, we have

$$\begin{aligned}
(f_1, f_3, f_5) \mapsto (& \\
&\text{if } \mathrm{c}_1 \text{ true} : T^{string}.C\_left.vn.add\,(ns)\,, \\
&\text{if } \mathrm{c}_2 \text{ true} : T^{string}.C\_right.vn.add\,(ns)\,, \\
&\text{if } \mathrm{c}_3 \text{ true} : T^{string}.C\_right.vn.add\,(ns)\,, \\
&\text{if } \mathrm{c}_4 \text{ true} : T^{string}.C\_right.vn.add\,(ns) \\
&)
\end{aligned} \tag{4}$$

which is saved in a $3 \times 3 \times 3$ matrix.

With the help of this processing and mapping we can easily parallelize checkings, by parting over the two given nodes. Since, in every matrix position the (c_1, c_2, c_3, c_4) tuple is already clearly given, we can already prepocess instead of equation 4 like in the following way, by saving the left and write tuples as functions of `ns` pointer triples with

$$\begin{aligned}
(f_1, f_3, f_5) \mapsto (& \\
&T^{string}.C\_left.vn.add\,((\text{if } \mathrm{c}_1 \text{ true} : ns))\,, \\
&T^{string}.C\_right.vn.add\,((\text{if } \mathrm{c}_1 \text{ true} : ns, \text{if } \mathrm{c}_2 \text{ true} : ns, \text{if } \mathrm{c}_3 \text{ true} : ns)))
\end{aligned} \tag{5}$$

With this, we are now able to reduce the complexity. At first, we take the value of the flag triple (f_1, f_3, f_5) to derive the matrix entry 5, which two entries (the one for $T^{string}.C\_left$ and the one for $T^{string}.C\_right$ can be run and saved in parallel.

If we look at our second case with $\mathcal{C}^{\alpha}_{[q]} := (0, 3)$ and $\mathcal{C}^{\beta}_{[q]} := (2, 1)$, we see that we can do the same thing for this one.

1. Case $\mathcal{C}^{\alpha}_{[q]}\,(0, 3)$:

   (a) (f_2, f_4, f_6)

2. Case $\mathcal{C}^{\beta}_{[q]}\,(2, 1)$:

   (a) (f_1, f_3, f_6)

   (b) (f_1, f_4, f_5)

   (c) (f_2, f_3, f_5)

3.4 General `ns` check improvement

We want to discuss some possible data structure aspects for an improvement of all `ns` values at once checking.

We know that in deph b we can have $2^{1.58 \cdot b}$ `ns` values at each T^{string} node. For each of them we have to make in the the worst case one check for the left $\mathcal{C}$ rule and three checks for the right $\mathcal{C}$ value, hence totally worst case situation are $3 \cdot 2^{1.58 \cdot b}$ checks which have to be done sequentially in each deph of the binary tree T^{string}, regarding time complexity.

Now, we want to propose the following approach considering the checks for depth b:

Step 1 (Preprocessing for checks)**.** *Consider a preprocssing with the following assumptions:*

- *We assume that all necessary memory is already allocated with a negotiated fixed addressing scheme so that particular places in memory can directly be accessed respectively it can directly written at the specific position (for example like an array with allocated memory and empty entries). In further steps we will denote it simply as datastructure and talk about it like an array (although it not has to be mandatory an array!)*
- *We assume this allocated space is organized in two data structures. The data structure DS_T which saves the results of checks and flag in tree T and a data structure $DS_{T^{string}}$ which saves pointers to addresses of the needed node checks and flags, a function calling which calls the saving instructions, the savings itself of the solutions, and a final instruction for each valid solution for the determination of the next $\mathcal{C}$ with `ns` constellations and addressings, which are needed.*

Next, we go to tree T at depth b.

Step 2. *We have two interacting (= connected) datastructures.*

1. *We check at each node of T in depth b,*

   (a) $node.count == 0$
   (b) $node.count \geq 1$
   (c) $node.count \geq 2$
   (d) $node.count \geq 3$
   (e) $node.count == node.parent.count$

   and determine the flag

   (a) *If* $node.count == 0$ *is true:*
   Set flag $node.flag = 0$.
   (b) *Else if* $node.count \geq 1$ *is true and* $node.count! = node.parent.count$ *is true:*
   Set flag $node.flag = 1$.
   (c) *Else if* $node.count \geq 1$ *is true and* $node.count == node.parent.count$ *is true:*
   Set flag $node.flag = p$, *with* p *be a fixed prime number.*

   We save this results each immediately after determination as an ***`true`*** *or* ***`false`*** *entry in our tree T datastructure DS_T. For example, we can use here an array with one row for each kind of check and the flag and with one column for each node. So, we have an array with size* 6×2^b.

2. *We already discussed in the last section, that we just need three specific flags (we chose (f_1, f_3, f_5)) to be able to directly derive which of the four checks are true and which are false (the belonging (c_1, c_2, c_3, c_4) constellation). So we can define a general function depending on the input (f_1, f_3, f_5) and* ***`ns`****, just consisting of saving instructions. Depending on the specific value combination of the flags (f_1, f_3, f_5)*

it chooses if we have to save e.g. only the first check, or only the second check, or both checks, etc.... (Similiar to equation (4 respectively 5), but NOT as list saving like it is done there with the `.add(...)` *part. We will still come back to this, later).*

So, we assume that we have a datastructure $DS_{T^{string}}$ in which we have saved for each `ns` *of each node of T^{string} in depth b an addressing (e.g. pointers) to the necessary flags in DS_T which we need to know, to know which checks for* `ns` *become true and which become false. This means for one node of T^{string} we have e.g. an array of size $1 \times 2^{1.58 \cdot b}$ addressings of needed flags.*

Now, if we already determined all checks in T, after determining the flag f of each node, then in $DT_{T^{string}}$ the addressing pointers get the necessary information (which is $\mathcal{O}(1)$ for all `ns` *of the tree in depth b since this runs in parallel). If we also add a call instruction to each $DS_{T^{string}}$ entry, then we can immediately call our just discussed saving function, also in parallel for all* `ns` *as just one saving command with $\mathcal{O}(1)$ (since we use already allocated memory instead of lists for saving the* `ns` *values anymore), too.*

So, we see that we already checked and saved solutions for depth b in constant time. What we not discussed is the preprocessing part from the beginning. We look at this with the following, now.

1. Since, we use fixed sized memory structures, the allocation of the whole necessary memory space for all possible entries can be done in just one step totally (so we have time complexity for all allocation of $\mathcal{O}(1)$).

2. We have one general function (or better class method) $F((f_1, f_3, f_5), ns)$, which consists of the general, functional, saving instructions. Depending which (f_1, f_3, f_5) is given, it choose the right ones and saves it at the binary tree places of its belonging instance.

3. For this whole considerations, we need to know already in the preprocessing step the next $\mathcal{C}$ rules of the next step. We can do this if we already add in step 2 an additionally call after saving our solutions, which immediately determines the next rules and their belonging necessary `ns` constellation and node addressings, which can we do if memory space of data structure DS_T at this moment also already exists and is allocated, to know the necessary addresssing information. This can be done without problems, since DS_T only already have to be allocated but not already the entries determined and total allocation runs in time complexity $\mathcal{O}(1)$.

4. Since we use already allocated space with fixed addressing, we do the saving instructions in parallel. Alternatively, we could also define tuple savings like in equation (5) which also breaks the needed time complexity down to totally constant time complexity for saving all valid solutions of $\mathcal{O}(3)$ in depth b.

So, we can summarise a final step.

Step 3. *After saving all solutions in step 2 in parallel (the solutions itself are also saved in the data structure $DS_{T^{string}}$), we call an additional function, also in parallel, for each valid solution, directly after we saved this solution to determine the next $\mathcal{C}$ with* `ns` *constellations, derive the necessary addressings for our pointers in $DS_{T^{string}}$ already for the next step and save them in $DS_{T^{string}}$.*
In the meantime, the total allocation of the next step memory space of DS_T already happened with time complexity $\mathcal{O}(1)$.

With this steps we get a checking of all possibile checks for all valid `ns` solution in constant time in each depth step b. Hence, we get, instead of $U^{1.58 \cdot b}$ with $U := 2^b$ before, a time complexity of constant time $b+1$.

All this way of improvement only works, if we assume that all allocation, function calls and saving can be done in constant time and that the additional parallelization part, compared with the approach before, not considerably raise the computation time.

Conclusion

We see that we are able to improve the complexitiy under the assumption of specific constraint conditions.

One additional improvement (not presented here) consists in the possibility of a combination of both approaches (the original one in the paper, and the one here). We can split the whole set of solution checkings in such a way, that we get a worst case number of checks sequentially of $2 \cdot 2^b$ by putting the rest of checks in parallel like here presented, which can be done, if we use fixed allocated memory space like here presented, too.

Acknowledgement

Thanks to the private donators who financially support this work.

License