A 3SUM algorithm on the binary addition level of atomic generated real numbers

-

Discussion doc: checkValid ns values improvement

Carolin Zöbelein*

September 13, 2020

DRAFT

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

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:

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

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

$$\begin{aligned}(f_1, f_3, f_5) \mapsto (&\\ &\text{if } c_1 \text{ true} : T^{string}.C\_left.vn.add\,(ns)\,,\\ &\text{if } c_2 \text{ true} : T^{string}.C\_right.vn.add\,(ns)\,,\\ &\text{if } c_3 \text{ true} : T^{string}.C\_right.vn.add\,(ns)\,,\\ &\text{if } c_4 \text{ true} : T^{string}.C\_right.vn.add\,(ns)\\ )&\end{aligned} \tag{4}$$

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

$$\begin{aligned}(f_1, f_3, f_5) \mapsto (&\\ &T^{string}.C\_left.vn.add\,((\text{if } c_1 \text{ true} : ns))\,,\\ &T^{string}.C\_right.vn.add\,((\text{if } c_1 \text{ true} : ns, \text{if } c_2 \text{ true} : ns, \text{if } c_3 \text{ true} : ns))\end{aligned} \tag{5}$$

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 XXX

TODO

Conclusion

TODO

Acknowledgement

Thanks to the private donators who financially support this work.

License