The 3SUM problem admits subquadratic solutions

Roberto Borelli
borelli.roberto@spes.uniud.it

Abstract

In this work, I consider the 3SUM problem. Recent years' studies have shown that the problem admits a subquadratic solution. The 3SUM problem has been used in the area of fine-grained complexity to establish lower bounds to a wide range of other problems (which have shown to be 3SUM-hard) for example in the computational geometry area. In this paper, I examine the Freund approach to obtain a subquadratic algorithm. To obtain a saving in the complexity several tricks have been applied and in particular it has been shown how to efficiently enumerate the so-called chunks through a correspondence with paths in a matrix and then all pairs of blocks agreeing with such derived chunks are obtained through a reduction to the DOMINANCE-MERGE problem.

Contents

1 Preliminaries

In this section, I start examining the 3SUM problem. I define the main problem and I show how it is equivalent to some slightly different variants. I briefly present the cubic solution which is a brute force search and a simple quadratic solution which is obtained from the algorithm for ORDERED-2SUM. Then, I consider the simpler-looking problem CONVOLUTION-3SUM which turns out to be equivalent to the original problem. Finally, I consider the importance of the 3SUM problem in the context of fine-grained complexity and computational geometry, I present the related conjectures and the notion of 3SUM-hard problems.

1.1 The 3SUM problem and trivial solutions

1.1.1 Definitions

Let's start with the definition of the 3SUM problem.

Problem 1 (3SUM)**.** *Given lists $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ of n reals, find (a, b, c) with $a \in \mathcal{A}, b \in \mathcal{B}, c \in \mathcal{C}$ such that $a + b = c$.*

Is easy to see that the following two variants are equivalent to the original problem.

SUM-0-VARIANT Given lists $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ of n reals, find (a, b, c) with $a \in \mathcal{A}, b \in \mathcal{B}, c \in \mathcal{C}$ such that $a+b+c = 0$.

SINGLE-LIST-VARIANT Given a list $\mathcal{S}$ of n reals, find indexes $a, b, c \in \mathcal{S}$ such that $a + b + c = 0$.

Given an algorithm for 3SUM, and given input lists $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, to solve the SUM-0-VARIANT just invert the sign of every number in $\mathcal{C}$. The converse (from a SUM-0-VARIANT algorithm, to a 3SUM one) is done in a symmetric way. To solve the SINGLE-LIST-VARIANT given the input list $\mathcal{S}$, put $\mathcal{A} \leftarrow \mathcal{S}$, $\mathcal{B} \leftarrow \mathcal{S}$, and $\mathcal{C} \leftarrow -\mathcal{S}$ then run the 3SUM algorithm on input $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and if the algorithm returns the triple (a, b, c), the triple $(a, b, -c)$ is the solution for the SINGLE-LIST-VARIANT. The converse is a bit more complicated and is due to Gajentaan and Overmars [GO95]. Suppose we want to solve 3SUM on input $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$. We want to create a set S such that whenever three elements $x, y, z \in S$ sum to 0, namely $x + y + z = 0$, we can find $a \in A, b \in B, c \in C$ such that $a + b = c$. Let $m = 2 \max(\mathcal{A}, \mathcal{B}, \mathcal{C})$. We put elements in S with the following rules:

- $a \in \mathcal{A} \implies a' = a + m \in \mathcal{S} \quad a' \in (m, 1.5m]$
- $b \in \mathcal{B} \implies b' = b \in \mathcal{S} \quad b' \in (0, 0.5m]$
- $c \in \mathcal{C} \implies c' = -c - m \in \mathcal{S} \quad c' \in [-1.5m, -m)$

It follows that whenever $a + b = c$ we have $a' + b' + c' = 0$, furthermore, if we find three elements in S that sum up to 0 the three elements must belong to the three different sets. This last observation follows from the bounds on the values for elements a', b', and c'.

1.1.2 Trivial solutions

In the following box I show the very basic cubic solution for 3SUM, which is nothing more than a brute-force search.

```
1 forall a ∈ A, b ∈ B, c ∈ C
2       if a + b = c
3               return (a, b, c)
4 return (nil, nil, nil)
```

To obtain a quite better result we can first consider the simpler ORDERED-2SUM problem.

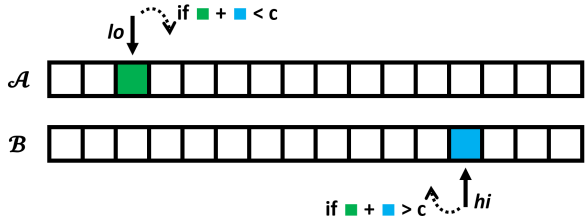


Figure 1: The key step in the linear time ORDERED-2SUM algorithm. If $\mathcal{A}[lo]+\mathcal{B}[hi] < c$ then $lo{+}{+}$, if $\mathcal{A}[lo] + \mathcal{B}[hi] > c$ then $hi - -$. If $\mathcal{A}[lo] + \mathcal{B}[hi] = c$ return the solution.

Problem 2 (ORDERED-2SUM)**.** *Given sorted lists $\mathcal{A}$, $\mathcal{B}$ of n reals, and a real c, find $a \in \mathcal{A}$ and $b \in \mathcal{B}$ such that $a + b = c$.*

The ORDERED-2SUM problem admits a simple linear-time solution. We consider two indexes: lo starting from the minimum of $\mathcal{A}$ (which is in position 1 since $\mathcal{A}$ is sorted) and hi starting from the maximum value of $\mathcal{B}$ which is in position n. At each step, we consider the quantity $q = \mathcal{A}[lo]+\mathcal{B}[hi]$. If q is equal to c we have found a solution for the problem. If $q < c$ we increase by one the index lo, otherwise we decrease by one the index hi. This process continues until lo reaches the maximum of $\mathcal{A}$ or hi reaches the minimum of $\mathcal{B}$. The key step of the algorithm can be represented as in figure 1. The maximum number of iterations taken by the algorithm is bounded by $2n$. The correctness of the algorithm is guaranteed by the invariant that each element $\mathcal{A}[i]$ with $i < lo$ and each element $\mathcal{B}[j]$ with $j > hi$ cannot be part of the solution. Before entering the cycle the invariant is trivially satisfied since there are no elements in $\mathcal{A}$ before index lo and there are no elements in $\mathcal{B}$ after index hi. Assuming that the invariant holds at the beginning of iteration $k-1$, it is easy to see that with the described rules the invariant continues to hold at the end of iteration $k-1$ and at the beginning of iteration k. To obtain a quadratic solution for the 3SUM problem is now sufficient to sort lists $\mathcal{A}$ and $\mathcal{B}$ and to invoke a linear number of times the linear algorithm for ORDERED-2SUM as described in the following box.

```
1 sort A, sort B
2 forall c ∈ C
3       (a, b, c) ← ORDERED-2SUM(A, B, c)
4       if (a, b, c) ≠ (nil, nil, nil)
5             return (a, b, c)
6 return (nil, nil, nil)
```

1.1.3 A simpler-looking problem

To obtain a $\mathcal{O}(n^2)$ algorithm for 3SUM, we can reduce the problem to an apparently simpler one called CONVOLUTION-3SUM.

Problem 3. *Given a list $\mathcal{A}$ of n reals, find indexes $i, j \in [n]$ such that $\mathcal{A}[i] + \mathcal{A}[j] = \mathcal{A}[i + j]$.*

Right from the definition of CONVOLUTION-3SUM, one can obtain a simple quadratic solution since there is only a quadratic number of possible tests that we might want to check. Even if this problem seems simpler, it turns out that CONVOLUTION-3SUM and 3SUM are inter-reducible. One direction is straightforward while the other requires a bit more technicalities and it is still a matter of research. In the reductions we refer to the SINGLE-LIST-VARIANT of 3-SUM.

Lemma 1. CONVOLUTION-3SUM *can be reduced to* 3SUM.

Proof. Suppose we want to solve an instance of CONVOLUTION-3SUM where $\mathcal{A}$ is the list of reals given as input. Construct a derived list of pairs $\mathcal{A}' = \{(\mathcal{A}[i], i) : i \in \mathcal{A}\}$. Define the addition on elements of $\mathcal{A}'$ as the pointwise addition $(\mathcal{A}[i], i) + (\mathcal{A}[j], j) = (\mathcal{A}[i] + \mathcal{A}[j], i + j)$. Furthermore, use the lexicographical order on $\mathcal{A}'$. Now we can solve 3SUM on the input list $\mathcal{A}'$. It is easy to see that $\mathcal{A}'[i] + \mathcal{A}'[j] = \mathcal{A}'[k] \iff (\mathcal{A}[i], i) + (\mathcal{A}[j], j) = (\mathcal{A}[k], k) \iff (\mathcal{A}[i] + \mathcal{A}[j] = \mathcal{A}[k]) \wedge (i + j = k) \iff \mathcal{A}[i] + \mathcal{A}[j] = \mathcal{A}[i + j]$.

□

Lemma 2. 3SUM *(on integers) can be reduced to* CONVOLUTION-3SUM *(on integers).*

Proof sketch. Suppose we want to solve an instance of 3SUM where $\mathcal{A}$ is the list of reals given as input. Suppose to have a map $h : \mathcal{A} \to [n]$ which is injective and linear in the sense that $h(x) + h(y) = h(x + y)$. We can construct a new list $\mathcal{A}'$ such that each element $x \in \mathcal{A}$ is placed in $\mathcal{A}'[h(x)]$. We have that $\exists x, y, z \in \mathcal{A} : x + y = z \implies \mathcal{A}'[h(x)] + \mathcal{A}'[h(y)] = \mathcal{A}'[h(x) + h(y)]$ since $h(z) = h(x + y) = h(x) + h(y)$ and so the CONVOLUTION-3SUM algorithm will find the solution. Furthermore, any $\mathcal{A}'[i] + \mathcal{A}'[j] = \mathcal{A}'[k]$ is a valid answer to 3SUM. Unfortunately, such a map h does not exist. Patrascu [Pat10] shows that if 3SUM requires $\Omega(n^2/f(n))$ expected time, CONVOLUTION-3SUM requires $\Omega(n^2/f^2(n \cdot f(n)))$ expected time. He uses a randomized reduction using a family of almost linear hash functions. He considers $h(x) = ((a \cdot x) \mod 2^w)/2^{w-s}$ where a is a random odd number, w is the word size and the values produced are in the range $\{0, \ldots, 2^s - 1\}$. Chan [CH20] improved the result showing that if we can solve CONVOLUTION-3SUM in *deterministic* $\Omega(n^{2-\epsilon})$ time (with $\epsilon > 0$), we can solve 3SUM in *deterministic* $\Omega(n^{2-(\epsilon/5+\epsilon)})$ time. □

1.2 3SUM in fine-grained complexity

Complexity theory allows us to put problems in complexity classes. If a problem L is in a complexity class C, then we know that there exists a Turing machine that works on the limit imposed by C which accepts L. The hierarchy theorem says that if $f(n) \geq n$ is a proper function of complexity, $TIME(f(n))$, the class of problems that can be computed in deterministic time $f(n)$ is strictly included in $TIME(f^3(2n+1))$. This theorem suggests that we can try to distinguish problems inside the class P and have a finer notion of complexity. In particular, we can distinguish problems that are solvable in linear time, problems that are solvable in quadratic time but not solvable in $\mathcal{O}(n^{2-\epsilon})$ for $\epsilon > 0$ and so on. The classic notion of polynomial time reduction cannot be applied in this context, instead, in the area of fine-grained complexity the notion of *fine-grained reduction* is used. The approach taken in this area mimics classic concepts of computational complexity [Wil18], in particular, *fine-grained* hardness hypotheses are used to prove conditional lower bounds on problems using fine-grained reductions.

The 3SUM problem is in some sense *hard* to solve efficiently, and the related (widely believed) conjecture was the following:

Conjecture 1 (3SUM hard conjecture)**.** 3SUM *cannot be solved in subquadratic time.*

This hard conjecture implies that the lower bound for 3SUM is $\Omega(n^2)$. This problem became central in fine-grained complexity, in fact in 1995, Gajentaan and Overmars [GO95] introduced for the first time the notion of 3SUM-hard problems and they proved conditional lower bounds for many computational geometry problems.

Definition 1 (3SUM-hard (definition by Gajentaan and Overmars))**.** *We call a problem P* 3SUM*-hard if and only if every instance of* 3SUM *of size n can be solved using a constant number of instances of P of at most linear size and $o(n^2)$ additional time.*

From the definition, it immediately follows that if P is 3SUM-hard, it is impossible to find a subquadratic algorithm for P unless the 3SUM hard conjecture is false. Surprisingly, Gronlund and Pettie [GP14] recently refused the 3SUM hard hypothesis, in fact they exhibited the first subquadratic algorithm for 3SUM. This result gave the hope that subquadratic algorithms might exist for the geometric problems presented by Gajentaan and Overmars. Even if the hard conjecture has been refused, there exists a weaker conjecture which is still untouched nowadays.

Conjecture 2 (3SUM weak conjecture)**.** 3SUM *cannot be solved in time* $O(n^{2-\epsilon})$ *for* $\epsilon > 0$ *even by a randomized algorithm.*

Since the discovery of Gronlund and Pettie, the research was very active in this field, in particular:

[GP14] In 2014, Gronlund and Pettie proposed the first subquadratic algorithm with time complexity $\mathcal{O}\left(n^2 \left(\frac{\log\log n}{\log n}\right)^{2/3}\right)$

[Fre17] In 2017, Freund refined complexity to $\mathcal{O}\left(n^2 \left(\frac{\log\log n}{\log n}\right)\right)$

[Cha19] In 2019, Chan gives a further improvement by about one more logarithmic factor, obtaining time complexity $\mathcal{O}\left(n^2 \left(\frac{(\log\log n)^{\mathcal{O}(1)}}{\log^2 n}\right)\right)$

It is interesting to notice that each of the above algorithms, at some point use a computational geometry problem and so the relation with 3SUM and computational geometry is in some sense double. In computation geometry the notion of 3SUM-hard problem is used to prove conditional lower bounds, and conversely computational geometry problems have been used to find subquadratic algorithms for 3SUM. In the next section I describe in detail the deterministic algorithm proposed by Freund which uses the DOMINANCE-MERGE problem.

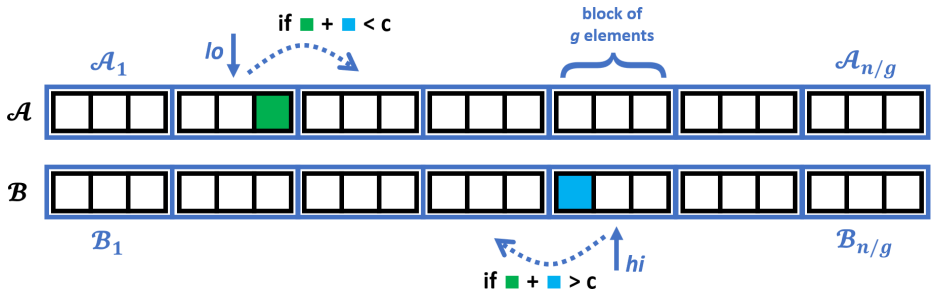


Figure 2: The key step in the subquadratic 3SUM algorithm. If $\max(\mathcal{A}_{lo}) + \min(\mathcal{B}_{hi}) < c$ then $lo{+}{+}$, if $\max(\mathcal{A}_{lo}) + \min(\mathcal{B}_{hi}) > c$ then $hi - -$. If c is found in $\mathcal{A}_{lo} + \mathcal{B}_{hi}$, return the solution.

2 The Freund approach

In this section I present the algorithm proposed by Freund to obtain a subquadratic running time for 3SUM. The algorithm is similar to the one proposed by Gronlund and Pettie but is slightly more efficient. To summarize the algorithm we can say that it resembles the linear time ORDERED-2SUM algorithm, but the input lists are divided in blocks. For each pair of blocks we need to calculate its ranking, and to do it efficiently we split rankings into chunks. The trick is that chunks can be shared through multiple rankings. To compute chunks, we have to avoid an exhaustive enumeration and this is also permitted by a reduction to the DOMINANCE-MERGE problem.

2.1 The algorithm

The structure of the algorithm is the following (red parts are going to be specified later).

1 Sort $\mathcal{A}$ and $\mathcal{B}$ and split them into n/g blocks of g elements

2 Preprocess $\mathcal{A}$ and $\mathcal{B}$

3 For each element $c \in \mathcal{C}$

- 3a Initialize $lo \leftarrow 0$ and $hi \leftarrow \frac{n}{g}$
- 3b Check if c is present in the sorted array $\mathcal{A}_{lo} + \mathcal{B}_{hi}$ of g^2 elements
- 3c If such c does not exist, update lo and hi and go to 3b

First $\mathcal{A}$ and $\mathcal{B}$ are sorted and split into blocks $\mathcal{A}_1, \ldots, \mathcal{A}_{n/g}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_{n/g}$ respectively. Denote by $\mathcal{A}_i + \mathcal{B}_j$ the multiset $\{\mathcal{A}_i[h] + \mathcal{B}_j[k] : h, k \in [g]\}$. Instructions 3b and 3c mimic the 2SUM algorithm. The key step is visualized in figure 2. We first check the presence of c in $\mathcal{A}_{lo} + \mathcal{B}_{hi}$. If it is present then we return. If c is smaller than $\max(\mathcal{A}_{lo}) + \min(\mathcal{B}_{hi})$ we increase by one the index lo, otherwise we decrease by one the index hi and we repeat the process until hi becomes lower than 0 or lo becomes greater than n/g. The correctness of this step follows from the invariant the members in blocks $\mathcal{A}_p$ with $p < lo$ and members in blocks $\mathcal{B}_p$ with $q > hi$ cannot participate to the solution. Intuitively the preprocessing (instruction 2) is going to create and sort each array $\mathcal{A}_i + \mathcal{B}_j$ and the check (instruction 3b) is a binary search. Overall the algorithm takes $\mathcal{O}\left(n \log n + T_{\text{preprocessing}} + n^2 \frac{\log g}{g}\right)$ time, since for each c (there are n possible values) the loop through instructions 3b and 3c is executed at maximum $2n/g$ times and each step involves a binary search through an array of size g^2 and hence the total time for instruction 3, 3a, 3b, and 3c is bounded by $\mathcal{O}\left(n^2 \frac{\log g}{g}\right)$. The challenge to obtain a subquadratic algorithm reduces to obtain a subquadratic preprocessing. Of course we cannot compute directly each $\mathcal{A}_i + \mathcal{B}_j$ since there are n^2/g^2 such arrays and each one is of size g^2. We will

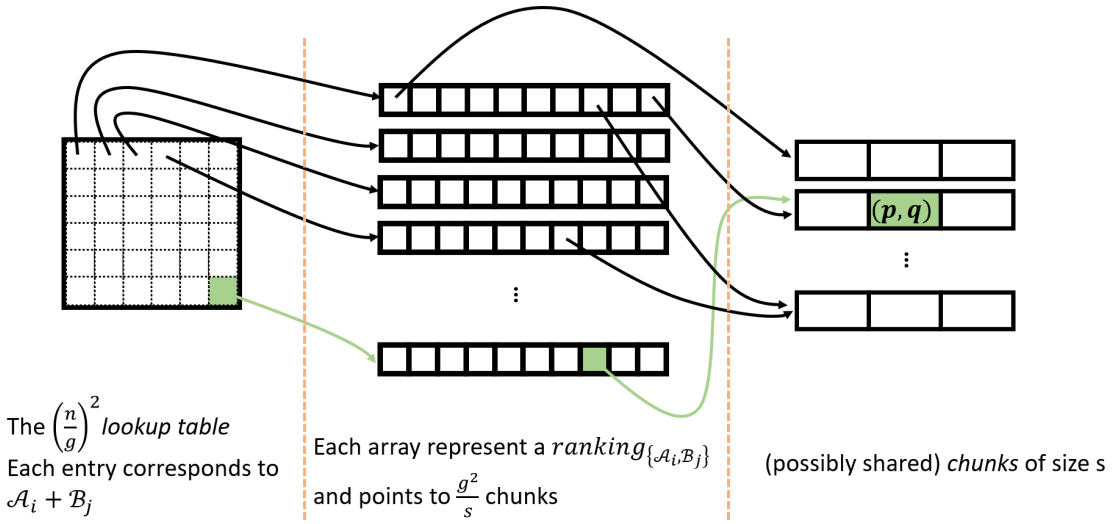


Figure 3: The triple indirect data structure. On the left we see the first layer which is the lookup table. At the center we see the layer of RK arrays and on the right the layer of chunk arrays.

introduce a triple indirect data structure such that 1) we can access the k-th position of the sorted array $\mathcal{A}_i + \mathcal{B}_j$ in constant time (this requirement is fundamental to obtain logarithmic time in the binary search) and 2) the preprocessing is subquadratic.

2.2 The triple indirect data structure

First, I introduce the notion of ranking. Let A and B be two particular blocks of lists $\mathcal{A}$ and $\mathcal{B}$, $ranking_{A,B}$ is an ordered sequence of g^2 pairs of type $(i,j) : i,j \in [g]$. Denote by $(i,j) \prec_{A,B} (i',j')$ the fact that the pair (i,j) comes before the pair (i',j') in $ranking_{A,B}$. The value of a pair (i,j) with respect to blocks A and B is the sum of the i-th element of block A and the j-th element of block B, namely $val_{A,B}(i,j) = A[i] + B[j]$. Assume that for fixed A and B the value $val_{A,B}(i,j)$ is unique. This assumption will be relaxed in section 2.4. The sequence $ranking_{A,B}$ contains all the pairs $(i,j) : i,j \in [g]$ ordered by their values, namely $(i,j) \prec_{A,B} (i',j') \iff val_{A,B}(i,j) < val_{A,B}(i',j')$. In other terms, consider the ordered list $A + B$, then to access $(A + B)[k]$ is sufficient to access $ranking_{A,B}[k]$ and obtain the pair (i_k, j_k) and then access to the respective positions in A and B, namely we have $(A+B)[k] = (A[i_k]+B[j_k]) = val_{A,B}(i_k, j_k)$. Of course we cannot compute explicitly each sequence $ranking_{A,B}$ for the same reason we could not compute each sorted array $\mathcal{A}_i + \mathcal{B}_j$ as pointed out in section 2.1. To overcome this problem, split each $ranking_{A,B}$ into g^2/s chunks of size s. The point in chunks, is that they can be shared across multiple rankings and hopefully this sharing will allow a subquadratic preprocessing. The challenge of computing chunks and allowing the sharing across rankings is going to be explained in section 2.3. For now, suppose that in some way, the computation of chunks is efficient and can be done in subquadratic time and space. In this section we center our focus on how the data structure built in the preprocessing and used in the binary search is organized. Having computed chunks, the goal of the data structure is to access at the k-th position of the sorted list $\mathcal{A}_i + \mathcal{B}_j$ in constant time. The data structure is visualized in figure 3 and it is composed of 3 levels. The first level is the lookup table L which has n/g columns (one per each block of $\mathcal{A}$) and n/g rows (one per each block of $\mathcal{B}$). Each entry $L[i,j]$ points to the array $RK_{\mathcal{A}_i,\mathcal{B}_j}$. The array $RK_{A,B}$ stores pointers to chunks such that if we concatenate all chunks pointed by $RK_{A,B}$ we obtain $ranking_{A,B}$. The second level is the one of RK arrays. There are n^2/g^2 such arrays (one for each entry of L) and each of these arrays is of length g^2/s since each ranking is split into g^2/s chunks. The third level is the level of chunk arrays. We denote one of these arrays by CA_T and each of these arrays has size s. Again, the whole point is that we don't need to compute n^2/s such arrays, otherwise we would obtain no gain in performance. The crucial point, which is going to be detailed later, is how to enable sharing of chunks to finally have subquadratic time in the

preprocessing. Is now easy to see that we can access to $(\mathcal{A}_i + \mathcal{B}_j)[k]$ in constant time as summarized in the following box.

1 Obtain from the entry $L[i,j]$ of the lookup table the pointer to the respective $RK_{\mathcal{A}_i,\mathcal{B}_j}$ array.

2 Consult $RK_{\mathcal{A}_i,\mathcal{B}_j}[k/s]$ to obtain the pointer to the chunk CA_T which stores the part of $ranking_{\mathcal{A}_i,\mathcal{B}_j}$ of interest.

3 Consult $CA_T[((k-1) \bmod s)+1]$ and obtain the pair (p,q).

4 Return the value $val_{\mathcal{A}_i,\mathcal{B}_j}(p,q) = \mathcal{A}_i[p] + \mathcal{B}_j[q]$

The time and space taken to allocate the lookup table L, to create RK arrays, to point L's entries to them, and to point RK arrays' entries to chunks is $\mathcal{O}\left(\frac{n^2}{g^2}\frac{g^2}{s}\right) = \mathcal{O}\left(\frac{n^2}{s}\right)$. This bound of course does not take into consideration the time taken to compute chunk arrays (which will be examined in section 2.3.5). Let's now explain in detail how chunk arrays can be efficiently computed.

2.3 Computing chunks

As said before, the point of chucks is that they may be shared across multiple rankings. The size of chunks s should be small such that it is more convenient to enumerate them (properly) rather than to compute them explicitly. I will first describe a naive exhaustive enumeration that doesn't take advantage of chunk sharing. Then, by reversing the order of operations, I will present the exhaustive enumeration with chunk sharing. The exhaustive enumeration is of course too expensive and we will see that a correspondence through special paths will allow us to prune the enumeration and obtain the desired complexity. Finally, to obtain all pairs of blocks that agree with the enumerated chunks we will use the so-called Fredman's trick and reduce the problem to the DOMINANCE-MERGE problem which admits an efficient solution.

2.3.1 Exhaustive enumeration

The first exhaustive enumeration one can think of, is the following (again, red parts are going to be specified later).

1 Forall pairs A, B

2 Forall $e : 1 \leq e \leq g^2/s$

3 For every $S : |S| = s, S \subset [g]^2$

4 For every permutation π

5 If S_π agrees with the e-th chunk of $ranking_{A,B}$

6 $CA \leftarrow$ NEW-CHUNK-ARRAY(S_π)

7 $RK_{A,B}[e] = CA$

At instruction 1 we enumerate all the possible blocks A and B, at instruction 2, all the possible chunk numbers. At instruction 3 we enumerate all the possible chunk sets and at instruction 4 we choose a permutation (from the $s!$ possibilities). The notation S_π used at instruction 5, represents

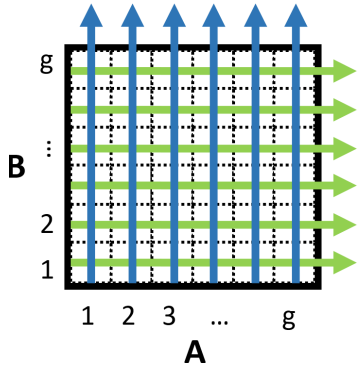


Figure 4: $ranking_{A,B}$ can be represented as a matrix of sorted rows and sorted columns. The value in the cell (i, j) is $val_{A,B}(i, j)$. The value $val_{A,B}(1, 1)$ is the minimum value in $A+B$ and $val_{A,B}(g, g)$ is the maximum value in $A + B$.

the list with all and only elements of S ordered following π. Despite the fact that this enumeration is too expensive, the conceptual problem is that it doesn't use chunk sharing. To solve this problem is sufficient to swap the order of things. First, enumerate all possible values for chunks, and for each enumerated chunk search for all the pairs of blocks that agree with that chunk. In the following box, this idea is formalized.

1 For every $S : |S| = s, S \subset [g]^2$

2 For every permutation π

3 Forall $e : 1 \leq e \leq g^2/s$

4 $\quad CA \leftarrow$ NEW-CHUNK-ARRAY(S_π)

5 $\quad$ Find all $A, B : (S, \pi, e)$ agrees with the e-th chunk of $ranking_{A,B}$

6 $\quad$ Forall such pairs A, B

7 $\quad\quad RK_{A,B}[e] = CA$

The main problem of this enumeration is that it is too expensive. The number of possible choices for S (instruction 1) is $\binom{g^2}{s}$. First, notice that if we choose $s = 1$ (or if s is a constant), the enumeration is efficient since there are only g^2 possible values for S, and for each S there is only one permutation, furthermore, the degree of chunk sharing is maximum. But this is not a solution since the cost to create the triple indirect data structure is $\mathcal{O}(n^2/s)$ as pointed out in section 2.2 and in total the preprocessing would be quadratic. On the other side, if we choose $s = g^2$, there is only one possible value for S, but then we will have $g^2!$ permutation (instruction 2) which is infeasible and also the degree of chunk sharing is minimum. Recall that $1 \leq s \leq g^2$. We would like to choose s as small as possible, in order to have a high rate of chunk sharing and to have few permutations to enumerate, but also sufficiently large in order to enable the subquadratic preprocessing. With such values of s the presented enumeration is too expensive and in the following I point out some problems. First, notice that for blocks A and B, $ranking_{A,B}$ can be represented by a $g \times g$ matrix with sorted rows and sorted columns (since both A and B) are (see figure 4). Regardless of blocks A, B certain pairs (S, e) will never agree with the e-th chunk of any $ranking_{A,B}$. For example the pair (g, g) will never be found in the first chunk (unless $s = g^2$), neither in the second and so on. The pair (g, g) will for sure be found in the last chunk of every $ranking_{A,B}$ since $val_{A,B}(g, g)$ is maximum in $A + B$ independently from blocks A, B. Similarly the pair $(1, 1)$ will be for sure be

found in the first chunk of $ranking_{A,B}$. So the first problem is that there are enumerated pairs (S, e) such that S contains the cell (i, j) which can't be found in the e-th chunk for any pair of blocks A, B. A second problem is that certain pairs can't be found in the same chunk regardless of the value of e and regardless of the blocks A and B. For example (unless the chunk size is g^2) we can't find the pairs $(1, 1)$ and (g, g) in the same chunk. Suppose by contradiction that the e-th chunk of $ranking_{A,B}(i, j)$ contains the pairs $(1, 1)$ and (g, g), then the chunk $e - 1$ should contain pairs whose value is less than $val_{A,B}(1, 1)$ but that's not possible since $val_{A,B}(1, 1)$ is minimum in ranking $A + B$. Similarly, the chunk $e + 1$ should contain pairs whose value is greater than $val_{A,B}(g, g)$ but that's not possible since $val_{A,B}(g, g)$ is maximum in ranking $A + B$. The second problem is that the enumeration goes through sets S that contain a subset $R \subseteq S$ that can't be found in any chunk. To conclude, this enumeration is problematic since most of the pairs (S, e) will never agree with any pair of blocks A, B. Our goal is to *prune* somehow the enumeration and to enumerate only *good* candidates (S, e). I call a pair (S, e) (where S is a set of size s and e is an integer $1 \leq e \leq g^2$) *good* if there exists a pair of sets A and B, each of g numbers, such that S is the set of the e-th chunk of $ranking_{A,B}$.

2.3.2 Avoiding exhaustive enumeration

To avoid an exhaustive enumeration as the one presented in section 2.3.1 we will have to be careful in identifying only the possible sets S that actually can be the e-th chunk of $ranking_{A,B}$ for values of A and B. Recall that a $ranking_{A,B}$ can be viewed as a $g \times g$ matrix as in figure 4. Instead of enumerating sets, we will enumerate pairs of paths on this matrix that are in correspondence to possible sets S and chunk numbers e. Let's call $M_{A,B}$ the matrix which represents $ranking_{A,B}$. We define a path on $M_{A,B}$.

Definition 2 (Path)**.** *A path on $M_{A,B}$ starting from the top left corner is a sequence of at most $2g - 1$ moves in the set $\{Right, Down\}$ which ends falling off the right edge or bottom edge. If x is the last instruction, there are exactly g instructions labeled with x in the path.*

If $g = 4$, $DDRD$ is not a possible path since the last instruction is D and there are only three D instructions. Consider now $M_{A,B}$ and a pair (i, j), we define a special path called $contour^{i,j}_{A,B}$.

Definition 3 (Contour)**.** *The path obtained by the following rules is called $contour^{i,j}_{A,B}$.*

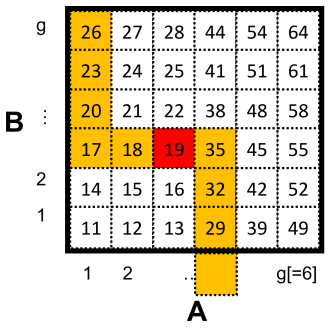
- *Start from the top left corner (cell $(1, g)$)*
- *If the value of the current cell is greater than $val_{A,B}(i, j)$, go down.*
- *If the value of the current cell is less or equal to $val_{A,B}(i, j)$, go right.*
- *Stop when you fall off the right edge or the bottom edge.*

The cell (i, j) is called the anchor of $contour^{i,j}_{A,B}$.

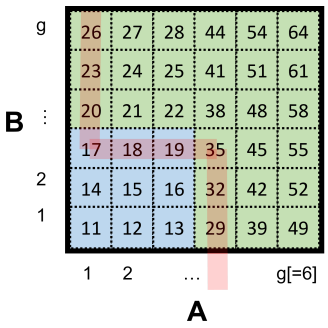
If we look closer, a $contour^{i,j}_{A,B}$ is nothing more than a run of the linear time algorithm for ORDERED-2SUM where the input arrays are A and B and we search for the value $val_{A,B}(i, j)$. The only difference between the linear time ORDERED-2SUM algorithm and the contour is that the contour doesn't stop when we reach the desired value. Note that a contour can be determined in $\mathcal{O}(g)$ time. In figure 5(a) I show an example of contour. Recall that all squares' values are unique and note that each contour must visit its anchor. The contour anchored in (i, j) separates in the matrix $M_{A,B}$ values greater than $val_{A,B}(i, j)$ (which are found above the contour) from values smaller or equal than $val_{A,B}(i, j)$ (which are found below the contour). Let's formally define the notion of a cell below the contour.

Definition 4 (Cell below the contour)**.** *The cell (h, k) is below $contour^{i,j}_{A,B}$ if there exists $k' \geq k$ such that $contour^{i,j}_{A,B}$ moves right from (h, k'). Otherwise (h, k) is above the contour.*

To make the definition clear, in figure 5(b), cells below $contour^{3,3}_{A,B}$ are blue-evidenced. The significance of the definition of cells below the contour is justified by the following property.



(a) $contour^{3,3}_{A,B}$. As an example, from $(1, g)$ we move down since $val_{A,B}(1, g) = 26 > 19 = val_{A,B}(3, 3)$.



(b) Cells below (blue color) and above (green color) $contour^{3,3}_{A,B}$. As an example, cell $(2, 1)$ is below the contour since from cell $(2, 3)$ the contour moves right.

Figure 5: The notion of contour and cells below the contour. In the example $A = \{2, 3, 4, 20, 30, 40\}$ and $B = \{9, 12, 15, 18, 21, 24\}$. The contour anchored in the cell $(3, 3)$ is the sequence $(D, D, D, R, R, R, D, D, D)$. The number of cells under the contour is 9, so $(3, 3)$ appears in the ninth position of $ranking_{A,B}$.

Property 1. *If the cell (h, k) is below $contour^{i,j}_{A,B}$, then $val_{A,B}(h, k) \leq val_{A,B}(i, j)$. If the cell (h, k) is above $contour^{i,j}_{A,B}$, then $val_{A,B}(h, k) > val_{A,B}(i, j)$.*

From the previous property it is clear that the position in $ranking_{A,B}$ of the pair (i, j) is the number of cells below $contour^{i,j}_{A,B}$. Furthermore, the following properties hold.

Property 2. *For each index $1 \leq r \leq g^2$ there exists exactly one cell (i, j) such that there are precisely r cells below $contour^{i,j}_{A,B}$.*

Property 3. *The number of cells below $contour^{i,j}_{A,B}$ can be calculated in $\mathcal{O}(g)$ time summing the indexes k of squares (h, k) from which the contour moves right.*

To clarify the last property, consider the example in figure 5(a), the cells from which the contour moves right are $(1, 3)$, $(2, 3)$ and $(3, 3)$, and if we sum the row indexes we obtain that there are $3 + 3 + 3 = 9$ squares below the contour. From the previous properties we can now characterize the set S of the e-th chunk of $ranking_{A,B}$ in terms of contours.

Property 4. *The set S is the e-th chunk of $ranking_{A,B}$ if and only if S is the set of squares above $contour^{k,l}_{A,B}$ and below $contour^{k',l'}_{A,B}$ where (k, l) is such that below $contour^{k,l}_{A,B}$ there are exactly $(e-1)s$ cells and (k', l') is such that below $contour^{k',l'}_{A,B}$ there are exactly es cells. In the special case $e = 1$ such (k, l) does not exist and the set S is the set of squares below $contour^{k',l'}_{A,B}$.*

Definition 5 (Path that dominates another path)**.** *We say that the path P' dominates the path P if every square above P' is also above P.*

To make the previous definition and property clear, look at the example in figure 6. Let's now see how to refine the exhaustive enumeration algorithm, using property 4, to generate all and only *good* pairs (S, e). Observe now that if we take a pair of paths P and P' such that P' dominates P, call S the set of squares between P and P', if S has size s and below P there are $(e - 1)s$ squares, the pair (S, e) is a *good* pair. The following algorithm formalizes this idea of paths' enumeration.

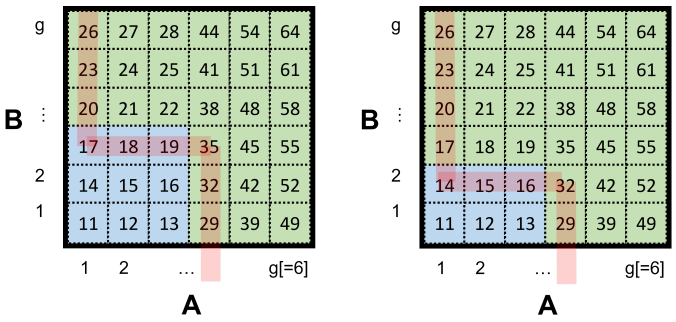


Figure 6: On the left we see $contour_{A,B}^{3,3}$ while on the right we see $contour_{A,B}^{3,2}$. Cells below the contour are blue-colored. Considering chunks of size $s = 3$, we see that under $contour_{A,B}^{3,3}$ there are $3 \times 3 = 9$ squares, under $contour_{A,B}^{3,2}$ there are $2 \times 3 = 6$ squares and hence the set $\{17, 18, 19\}$ is the set of the third chunk of $ranking_{A,B}$. Note that $contour_{A,B}^{3,3}$ dominates $contour_{A,B}^{3,2}$.

1 Forall pairs of paths P, P' s.t. P' dominates P

2 Check if between P and P' there are exactly s squares

3 Check if there exists e such that below P there are $(e-1)s$ squares

4 Let S be the set of squares between P and P'

5 Forall $(k, l), (k', l')$ possible anchors of P and P'

6 Forall permutations π such that (k', l') is the last element

7 $CA \leftarrow$ NEW-CHUNK-ARRAY(S_π)

8 Find all pairs $A, B : (S, \pi, P, P', (k, l), (k', l'))$ agrees with the e-th chunk of $ranking_{A,B}$

9 Forall such pairs A, B

10 $RK_{A,B}[e] = CA$

At instruction 1 we enumerate all pairs P and P' of paths such that the latter dominates the former. Instructions 2 and 3 are devoted to checking if the pair of paths is a good one (namely if the set of squares between them can be a candidate chunk set). Enumerating all the pairs of paths will not be a problem in complexity as we will see in section 2.3.5. At instruction 5 we enumerate all possible anchors for the paths. Recall that a contour must touch its anchor and, fixed a path, the only candidate positions for an anchor are the ones from which the path moves right. Now we have fixed the complexity of the enumeration but we still have to specify how to implement instruction 9, namely fixed a sextuple $(S, \pi, P, P', (k, l), (k', l'))$ (which identifies a possible e-th chunk), we want to find all the pairs of agreeing blocks. In other terms we want to find all pairs of blocks A, B such that property 4 is satisfied. Of course we cannot enumerate exhaustively all the pairs of blocks since they are n^2/g^2. To deal with this problem we will use a reduction to the so called DOMINANCE-MERGE problem. First in section 2.3.3 I define the problem and an efficient algorithm. Then in section 2.3.4 we will see how the DOMINANCE-MERGE problem will help us to implement efficiently instruction 9.

2.3.3 The DOMINANCE-MERGE problem

In this section I define the DOMINANCE-MERGE problem without touching the importance of it in our context. The use of this problem will be clear in section 2.3.4. Let's start by the definition of the problem.

Problem 4 (DOMINANCE-MERGE)**.** *Given N vectors in $\mathbb{R}^d$ such that each vector v has a color $c(v) \in \{red, blue\}$, find all pairs of vectors (a, b) such that $c(a) = red$, $c(b) = blue$ and $a < b$.*

The standard textbook algorithm for this problem, described by Shamos and Preparata [PS12] is the following. Call P the input list and for simplicity assume that no red point has the same k-th coordinate of any blue point, for every k.

1. If $|P| = 1$, stop
2. If $d = 0$, return all the pairs $(a, b) : a, b \in S, c(a) = red, c(b) = blue$
3. Find the median q of the projection of S into the d-th coordinate
4. Create lists $P_{red,<q}, P_{red,\geq q}, P_{blue,<q}, P_{blue,\geq q}$ where the list $P_{c,\sim q}$ is the list of c-colored points $p \in P$ such that if p_k is the k-th coordinate of the point p, it holds $p_k \sim q$
5. Solve DOMINANCE-MERGE$(P_{red,<q} \cup P_{blue,<q}, d)$
6. Solve DOMINANCE-MERGE$(P_{red,\geq q} \cup P_{blue,\geq q}, d)$
7. Solve DOMINANCE-MERGE$(P_{red,<q} \cup P_{blue,\geq q}, d-1)$

Analyzing the time taken by the algorithm, we have $T(N, d) = 2T(N/2, d) + T(N, d-1) + \mathcal{O}(N)$. Since $T(1, d) = \mathcal{O}(1)$, $T(N, 0) = \mathcal{O}(N)$ and each dominance pair is reported once, we can naively bind the time taken by the above algorithm by $\mathcal{O}(N \log^d N + |D|)$ where D is the set of emitted dominance pairs. This is the bound obtained by Preparata and Shamos [PS12]. Fixing a parameter b and letting $T'(R) = \max_{d,N:b^d N \leq R} T(N, d)$ Chan [Cha08] is able to establish a finer bound (on the same algorithm). In particular he proves that the algorithms runs in $\mathcal{O}(c_\epsilon^d N^{1+\epsilon} + |D|)$ for any constant $\epsilon \in (0, 1)$, where $c_\epsilon = 2^\epsilon/(2^\epsilon - 1)$. For our use in section 2.3.4, we put $\epsilon = 0.5$ and so $c_\epsilon < 4 = 2^2$ and we obtain the bound $\mathcal{O}(2^{2d} N^{1.5} + |D|)$.

2.3.4 The reduction to the DOMINANCE-MERGE problem

In this section I explain how to implement efficiently instruction 9 of the refined enumeration algorithm presented in section 2.3.2, using a reduction to the DOMINANCE-MERGE problem described in section 2.3.3. Given $(S, \pi, P, P', (k, l), (k', l'))$ we want to find all the pairs of agreeing blocks. First notice that by instructions 1, 2, and 3 we already know that (S, e) is a *good* pair[1]. The goal now is, to check if a fixed pair A, B satisfies the remaining conditions of property 4; in such a case we say A, B agrees with $(S, \pi, P, P', (k, l), (k', l'))$. First let's make clear what we mean by the term *agree*.

Definition 6 (Pair of blocks that agrees)**.** *We say that the pair of blocks A, B agrees with the sextuple $(S, \pi, P, P', (k, l), (k', l'))$ if 1) P is contour$^{k,l}_{A,B}$, 2) P' is contour$^{k',l'}_{A,B}$ and 3) S ordered by π is sorted in increasing order in terms of $val_{A,B}(\cdot)$.*

Recall that the sextuple $(S, \pi, P, P', (k, l), (k', l'))$ has been created by only looking at structural conditions of chunks following property 4, regardless of the actual values contained in blocks. Now we are interested to see which pairs of blocks, with their instantiated values, agree with such a sextuple.

We can translate the definition of A, B *agrees with* $(S, \pi, P, P', (k, l), (k', l'))$ into three sets of inequalities (one set for each point in the definition).

[1] That is, (S, e) is a candidate set for the set of the e-th chunk for some blocks A and B.

1. A set of $|P|-1$ inequalities, one for each cell touched by P (except for $(k.l)$).

$$\{A[i]+B[j] < A[k]+B[l] : P \text{ moves right from } (i,j)\} \cup$$
$$\{A[k]+B[l] < A[i]+B[j] : P \text{ moves down from } (i,j)\}$$

2. A set of $|P'|-1$ inequalities, one for each cell touched by P' (except for (k',l')).

$$\{A[i]+B[j] < A[k]+B[l] : P' \text{ moves right from } (i,j)\} \cup$$
$$\{A[k]+B[l] < A[i]+B[j] : P' \text{ moves down from } (i,j)\}$$

3. A set of to $s-1$ inequalities. Denote by (a_k, b_k) the pair $S_\pi[k]$.

$$\{A[a_k]+B[b_k] < A[a_{k+1}]+B[b_{k+1}] : 1 \le r < s\}$$

Notice that the first set is justified since for each cell $(i,j) \in P$ from which P moves right, if $val_{A,B}(i,j) < val_{A,B}(k,l)$ then $val_{A,B}(i,j') < val_{A,B}(k,l)$ for all $j' \le j$. Similarly, for each cell $(i,j) \in P$ from which P moves down, if $val_{A,B}(i,j) > val_{A,B}(k,l)$ then $val_{A,B}(i',j) > val_{A,B}(k,l)$ for all $i' \ge i$. For the second set, it holds a similar property. This is just another way of saying that if we want to check if P is $contour_{A,B}^{k,l}$, it suffices to check the relations of $val_{A,B}(i,j)$ with respect to $val_{A,B}(k,l)$ just for cells (i,j) along the contour. If those relations are satisfied, the relations between $val_{A,B}(k,l)$ and all the other values of cells not in P are satisfied since A, B are sorted.

The three sets of inequalities can be rewritten by moving all the A-terms to the right and all the B-terms to the left, for example $A[a_k]+B[b_k] < A[a_{k+1}]+B[b_{k+1}]$ becomes $B[b_k]-B[b_{k+1}] < A[a_{k+1}]-A[a_k]$. This step is called Fredman's trick and it is also used by Gronlund and Pettie [GP14] and Chan [Cha19]. Combining the three sets we can write the conditions that A, B *agrees with* $(S, \pi, P, P', (k,l), (k',l'))$ as a single vector inequality $v^B < v^A$ where v^B (resp. v^A) is a k-dimensional vector ($k = |P|+|P'|-3+s < 4g+s$) such that the u-th component is the left hand side (resp. right hand side) of the u-th inequality described above. The symbol "$<$" is to be interpreted pointwise. Now I define formally the problem of finding all the agreeing pairs of blocks.

Problem 5 (ALL-AGREEING-PAIRS (first formulation))**.** *Given $(S, \pi, P, P', (k,l), (k',l'))$, find all the pairs of blocks A, B such that A, B agrees with the sextuple, that is $v^B < v^A$.*

To solve the ALL-AGREEING-PAIRS problem the naive solution is to enumerate all pairs A, B and for each pair construct the vectors v^A, v^B and check if $v^B < v^A$. Of course that's to expensive since, as already observed, there are n^2/g^2 such pairs. The key observation is that even though there are n^2/g^2 such pairs, the vectors to construct are only $2n/g$ since by Fredman's trick v^A depends only on A and v^B depends only on B. Thus, the ALL-AGREEING-PAIRS problem can be reformulated as follows.

Problem 6 (ALL-AGREEING-PAIRS (second formulation))**.** *Given n/g vectors v^A and n/g vectors v^B find all pairs A, B such that $v^B < v^A$.*

In this second formulation it is clear that the ALL-AGREEING-PAIRS problem is nothing more than an instance of the DOMINANCE-MERGE problem described in section 2.3.3. In particular the input list of the DOMINANCE-MERGE problem is composed by $2n/g$ vectors. Vectors of type v^B are red-colored and vectors of type v^A are blue-colored. The input dimension d is bounded by $4g+s$ as seen above and so we can solve the ALL-AGREEING-PAIRS problem in time $\mathcal{O}\left(|D| + 2^{8g+2s}\left(\frac{2n}{g}\right)^{1,5}\right)$.

2.3.5 Complexity of chunks' computation

Now we have all the ingredients for the preprocessing of chunks. It remains to check the overall time spent in the enumeration algorithm. First let's count the number of pairs of paths P, P'. For a single path P, let's call $x \in \{D, R\}$ the type of the last instruction of P, and let $\bar{x}$ be the other instruction. A path of length $j \ge g$ is composed by g instructions x and $j-g$ instructions $\bar{x}$ and it is a permutation of the multiset $\left\{x^g, \bar{x}^{j-g}\right\}$. The number of possible paths P is:

$$2\sum_{j=0}^{g-1}\left(\frac{(j+g-1)!}{(g-1)!(j)!}\right)$$

which is bounded by $\mathcal{O}(2^{2g})$ (see lemma 4 of appendix A)and so the number of pairs of paths is bounded by $\mathcal{O}(2^{4g})$. Note that the check that P' dominates P can be done in $\mathcal{O}(g)$ time. The computation of the set of pairs S between P and P' is done in $\mathcal{O}(g+s)$ time. The number of possible anchors for P and P' is bounded by g^2. The number of permutations of S is $s! \leq 2^{s \log s}$ (see lemma 3 of appendix A). The cost to create a chunk is $\mathcal{O}(s)$, the cost for a call of the *dominance-merge* algorithm is $\mathcal{O}\left(|D| + 2^{8g+2s}\left(\frac{2n}{g}\right)^{1,5}\right)$ and the cost to point RK arrays to chunks is already been taken into consideration in section 2.2. In total we have:

$$\begin{aligned}
&\mathcal{O}\left(2^{4g}\left(g+s+\left(g^2 2^{s\log s}\left(s+|D|+2^{8g+2s}\left(\frac{2n}{g}\right)^{1,5}\right)\right)\right)\right)\\
&=\mathcal{O}\left(\tilde{D}+g^2 2^{12g+2s+s\log s}\left(\frac{2n}{g}\right)^{1,5}\right)
\end{aligned}$$

where $\tilde{D}$ is the total number of dominance pairs emitted. Since dominance pairs are in 1:1 correspondence with entries of arrays $RK_{A,B}$, this cost has already been considered in section 2.2. Finally notice that $2n/g < n$ and hence the total time spent on computing chunks is bounded by $\mathcal{O}\left(g^2 2^{12g+2s+s\log s} n^{1,5}\right)$.

2.4 Relaxing the uniqueness hypothesis

The only thing which remains to be fixed is to relax the uniqueness hypothesis made in section 2.2. For a pair of blocks we assumed that all values in $A+B$ are unique. Reviewing all the steps of the algorithm, one sees that the correctness of the preprocessing phase is guaranteed by the following properties.

1. The order of values in $A+B$, respects the order of appearance of pairs in $ranking_{A,B}$, that is, $val_{A,B}(i,j) < val_{A,B}(h,k) \implies (i,j) \prec_{A,B} (h,q)$.
2. Consider the matrix $M_{A,B}$ which represents $ranking_{A,B}$. We have that the order of values in $A+B$ is monotone in the vertical and horizontal directions. In other terms $i < h \implies (i,j) \prec_{A,B} (h,j)$ and $j < k \implies (i,j) \prec_{A,B} (i,k)$.
3. Given pairs (i,j) and (h,k), it is *easy* to decide the relation $(i,j) \prec_{A,B} (h,k)$ and the comparison can be decomposed into an A part and a B part.

The first property allows binary searches, the second allows the partitioning by contours and the third enables the efficient use of the DOMINANCE-MERGE algorithm. To relax the assumption, we can slightly modify the algorithm as long as we are able to maintain the three properties. We present two solutions, a first one proposed by Freund [Fre17] and a second one presented by Gronlund and Pettie [GP14]. In the end it turns out that the hypothesis was made just to simplify the language to describe the preprocessing. The Freund solution consists in redefining the order of appearance in the following way.

$$\begin{aligned}
(i,j) \prec (h,k) \iff &(val_{A,B}(i,j) < val_{A,B}(h,k)) \vee\\
&(val_{A,B}(i,j) = val_{A,B}(h,k) \wedge i < h) \vee\\
&(val_{A,B}(i,j) = val_{A,B}(h,k) \wedge i = h \wedge j < k)
\end{aligned}$$

It is easy to see that there are no equalities and the three properties are satisfied.

Gronlund and Pettie, propose to map elements $val_{A,B}(i,j)$ into a totally ordered universe U (with addition and subtraction) such that squares' values are unique. The algorithm is then redefined in

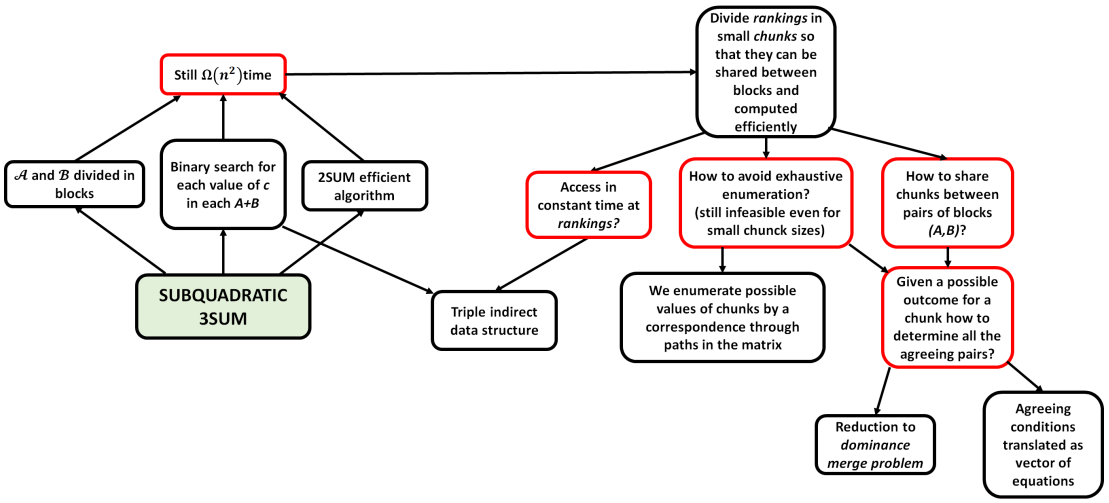


Figure 7: Final ingredients of the Freund algorithm. Red boxes represent problems while black boxes represent ideas or solutions.

terms of elements of U instead of numbers. In particular if we use $val_{A,B}(i,j) = A[i] + B[j] \longrightarrow (A[i] + B[j], i, j)$ together with pointwise addition and subtraction and the lexicographical order to compare elements of U, is again easy to see that the three properties are satisfied.

2.5 Overall complexity

In total, the overall time taken by the algorithm is composed by three parts:

1. The time for binary searches is bounded by $\mathcal{O}\left(n^2 \frac{\log g}{g}\right)$
2. The time to prepare the data structure is bounded by $\mathcal{O}\left(\frac{n^2}{s}\right)$
3. The time taken to compute chunks is bounded by $\mathcal{O}\left(g^2 2^{12g+2s+s\log s} n^{1,5}\right)$

By looking at points 1) and 2) we would want to put g and s as large as possible, but in this way the third bound increases. Furthermore from the first two bounds we see that the only reasonable choice for s is $s \in \Theta(g/\log g)$. If we put g above $\mathcal{O}(\log n)$ the third bound will be $n^{\omega(1)}$ which is unacceptable. If we put $g = \Theta(\log n)$ the third bound will be a polynomial whose degree is controllable by the choice of the constants. In particular if we put $s = g/\log g$ we have $s \leq s \log s \leq g$, and the third bound becomes $\mathcal{O}\left(g^2 2^{15g} n^{1,5}\right)$. Finally if we put $g = \frac{1}{31}\log n$ the first two bounds become $\mathcal{O}\left(n^2 \frac{\log\log n}{\log n}\right)$ and the third becomes $\mathcal{O}(n^{2-\frac{1}{62}} \log^2 n)$. Since $(n^{2-\frac{1}{62}} \log^2 n) \in \mathcal{O}\left(n^2 \frac{\log\log n}{\log n}\right)$ (see lemma 5 of appendix A), the total time taken by the Freund algorithm is $\mathcal{O}\left(n^2 \frac{\log\log n}{\log n}\right)$ and thus this is a proof that the 3SUM problem can be solved in subquadratic time.

2.6 Final ingredients

In figure 7 I show all the main final ingredients that we used to solve 3SUM in subquadratic time. We started our algorithm by dividing the input lists in blocks and we mimed the efficient ORDERED-2SUM algorithm taking blocks as cells. At each step we performed a binary search in the sorted array $A + B$. Unfortunately we cannot compute and sort each array $A + B$ otherwise we would consume too much time. For this reason, we introduced the notion of ranking (which is nothing more than a first indirection to the sorted array $A + B$) and we divided each ranking into small chunks. To simplify things we assumed that for blocks A and B, values in $A+B$ are unique. Chunks are justified since they can be shared across multiple rankings. We organized all the things in a triple indirect

data structure which allows to access $(A+B)[k]$ in constant time as required to maintain logarithmic time in binary searches. The task of finding a subquadratic algorithm for 3SUM reduces to the task of designing an efficient way to do the preprocessing phase and in particular to compute chunks. The exhaustive enumeration of chunks presents two problems: 1) it is too expensive, 2) it doesn't allow chunk sharing. The latter difficulty is solved just by rearranging the order of operations. The former problem is more involved: the enumeration generates candidate chunk sets that structurally violet the property of a chunk. To deal with this problem we saw a correspondence between pairs of paths in the ranking matrix and possible chunk sets. To allow chunk sharing we needed to compute efficiently all the pairs of blocks that agree with a particular chunk set. We translated the notion of *agreeing block* into a set of inequalities and we reduced the ALL-AGREEING-PAIRS problem to the DOMINANCE-MERGE problem which admits an efficient solution. The final issue we had to overcome was to relax the uniqueness hypothesis. This can be done in several ways, the simpler one is to redefine the order of appearance $\prec$ of pairs in rankings such that there are no equalities and the properties which guarantee the correctness of the preprocessing are still satisfied. We did the overall complexity calculation, and by choosing the right values for the block size and for the chunk size, we were able to make the preprocessing subquadratic while also maintaining the search step of the algorithm subquadratic.

3 Conclusions

In this paper we examined some results about the 3SUM problem. First I've analyzed the most common variants showing how they are essentially inter-reducible. We've examined trivial textbook algorithms and I presented the CONVOLUTION-3SUM problem showing that it is reducible to 3SUM, and working on the domain of integers, also the converse reduction holds. I put 3SUM in the context of fine-grained complexity and computational geometry. The widely believed hypothesis telling that 3SUM could not be solved in subquadratic time has been recently refused by Gronlund and Pettie. Since then, the research in this area has become more and more active and several subquadratic algorithms for 3SUM were presented. I deeply described the approach taken by Freund which can be decomposed into a preprocessing phase and a search phase. To make the preprocessing subquadratic, several techniques have been adopted and we also made use of the DOMINANCE-MERGE problem. Even though subquadratic algorithms exist, it seems to me that they could be really slow in practice compared with standard quadratic algorithms. For example, the Freund algorithm has a running time of $\mathcal{O}\left(n^2 \frac{\log\log n}{\log n}\right)$; taking $n = 1.000.000.000$ we have that $\frac{\log_2 \log_2 n}{\log_2 n} \simeq 0.163 < \frac{1}{6}$, taking $n = 1.000.000.000.000$ we have that $\frac{\log_2 \log_2 n}{\log_2 n} \simeq 0.133 < \frac{1}{7}$. Taking an input size of 1000 billions our algorithm is quadratic with a multiplying constant of about $1/7$ due to the $\frac{\log_2 \log_2 n}{\log_2 n}$ factor. Even without practical testing the Freund algorithm, I can assert, with a certain degree of trustworthiness, that the overhead added by the subquadratic preprocessing dominates the constant $\frac{1}{7}$. In particular, I suspect that constants hidden by the $\mathcal{O}$-notation are much bigger than 7 since there are many steps involved in the computation of chunks. As a matter of fact, the algorithm presented to solve the DOMINANCE-MERGE problem involves the computation of the median in linear time. This can be done with a linear time selection algorithm, but these types of algorithms have on their own an asymptotic constant typically much greater than one. For these reasons it seems to me that even the trivial quadratic algorithm presented at the beginning, in practice, is faster (with reasonable input sizes) than the subquadratic Freund algorithm. At the time of writing, I see the result of Gronlund and Pettie (and all successive works) as important in the theory of fine-grained complexity but less important with respect to the task of creating the fastest 3SUM algorithm. It is still unknown if there exists an algorithm to solve 3SUM with running time $\mathcal{O}(n^{2-\epsilon})$ with $\epsilon > 0$. Even though it is widely believed that this is not the case, I recall that even the $\Omega(n^2)$ lower bound for 3SUM has been widely believed for decades. However, if the 3SUM conjecture is false, and thus we can solve the problem in $\mathcal{O}(n^{2-\epsilon})$, I suspect that we are far from seeing such a result. Even thought subquadratic algorithms exist, I see that the preprocessing is not really able to take advantage of the structure of the problem. In particular, we are not able to organize the problem in an efficient data structure which then makes the solution obvious. For now we are just applying a series of precise tricks that at the end allow a subquadratic running time.

4 Bibliography

[CH20] T. M. Chan and Q. He. "Reducing 3SUM to convolution-3SUM". In: *Symposium on Simplicity in Algorithms*. SIAM. 2020, pp. 1–7.

[Cha08] T. M. Chan. "All-pairs shortest paths with real weights in O (n 3/log n) time". In: *Algorithmica* 50 (2008), pp. 236–243.

[Cha19] T. M. Chan. "More logarithmic-factor speedups for 3SUM,(median,+)-convolution, and some geometric 3SUM-hard problems". In: *ACM Transactions on Algorithms (TALG)* 16.1 (2019), pp. 1–23.

[Fre17] A. Freund. "Improved subquadratic 3SUM". In: *Algorithmica* 77 (2017), pp. 440–458.

[GO95] A. Gajentaan and M. H. Overmars. "On a class of O (n2) problems in computational geometry". In: *Computational geometry* 5.3 (1995), pp. 165–185.

[GP14] A. Grønlund and S. Pettie. "Threesomes, degenerates, and love triangles". In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. IEEE. 2014, pp. 621–630.

[Pat10] M. Patrascu. "Towards polynomial lower bounds for dynamic problems". In: *Proceedings of the forty-second ACM symposium on Theory of computing*. 2010, pp. 603–610.

[PS12] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[Wil18] V. V. Williams. "On some fine-grained questions in algorithms and complexity". In: *Proceedings of the international congress of mathematicians: Rio de janeiro 2018*. World Scientific. 2018, pp. 3447–3487.

A Proof of bounds

In this appendix I show the proofs of some bounds that I have used in the previous sections.

Lemma 3. $s! \leq 2^{s \log_2 s}$

Proof. $s! \leq s^s = 2^{\log_2 s^s} = 2^{s \log_2 s}$ □

Lemma 4. $2\sum_{j=0}^{g-1} \left(\frac{(j+g-1)!}{(g-1)!(j)!} \right) \in \mathcal{O}(2^{2g})$

Proof. We have to prove that $\lim_{g\to\infty} \frac{2\sum_{j=0}^{g-1}\left(\frac{(j+g-1)!}{(g-1)!(j)!} \right)}{2^{2g}} = c \in \mathbb{R}$
Let's first study the nominator. First the term $\frac{1}{(g-1)!}$ can be taken out of the sum since it doesn't depend on j. The term $\frac{(j+g-1)!}{j!}$ can be rewritten as

$$\frac{(g-1+j)(g-2+j)(g-3+j)(g-4+j)\dots(g-(g-1)+j)(j)(j-1)\dots(2)(1)}{(j)(j-1)\dots(2)(1)}$$
$$= (g-1+j)(g-2+j)\dots(1+j)$$

Now the sum $\sum_{j=0}^{g-1} \Big((g-1+j)(g-2+j)\dots(1+j) \Big)$ can be expanded as

$$\Big((2g-2)(2g-1)\dots(g+1)(g) \Big) \left(1 + \frac{g-1}{2g-2} + \frac{(g-1)(g-2)}{(2g-2)(2g-3)} + \frac{(g-1)(g-2)(g-3)}{(2g-2)(2g-3)(2g-4)} + \right.$$
$$\left. + \dots + \frac{(g-1)(g-2)\dots(2)(1)}{(2g-2)(2g-3)\dots(2g-g)} \right)$$

where $\Big((2g-2)(2g-1)\dots(g+1)(g) \Big)$ can be rewritten as $\frac{(2g-2)!}{(g-1)!}$. Combining the rewritings and the sum expansion, the entire limit can be rewritten as:

$$\lim_{g\to\infty} \frac{1}{2^{2g}} \frac{2(2g-2)! \left(1 + \frac{g-1}{2g-2} + \frac{(g-1)(g-2)}{(2g-2)(2g-3)} + \dots + \frac{(g-1)!}{(2g-2)!/(g-1)!} \right)}{(g-1)!(g-1)!}$$

When g goes to infinity, the term $\left(1 + \frac{g-1}{2g-2} + \frac{(g-1)(g-2)}{(2g-2)(2g-3)} + \dots + \frac{(g-1)!}{(2g-2)!/(g-1)!} \right)$ tends to 2 since the contributions are $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots$, and hence the limit becomes

$$4 \lim_{g\to\infty} \frac{(2g-2)!}{(g-1)!(g-1)!(2^{2g})} = 4 \lim_{g\to\infty} \frac{(2g-2)(2g-3)\dots(g+1)(g)}{(g-1)!(2^{2g})}$$

in the last limit, notice that the nominator has $(g-1)$ terms. Let's examine the term $(g-1)!$ at the denominator. Since $(g-1)! = (g-1)(g-2)\dots(2)(1)$ we know that there are $(g-1) - \lfloor \frac{g-1}{2} \rfloor$ factors greater than $\frac{g-1}{2}$ and there are less than $\lfloor \frac{g-1}{2} \rfloor$ factors smaller than $\frac{g-1}{2}$. By taking out $\lfloor \frac{g-1}{2} \rfloor$ factors of value $\frac{g-1}{4}$ from the first half, and multiplying those factors on the second half we see that we can rewrite $(g-1)!$ as a product of $g-1$ factors all greater or equal than $\frac{g-1}{4}$, in other terms, we have $\left(\frac{x}{4}\right)^x \in \mathcal{O}(x!)$. Following this reasoning one can obtain $\left(\frac{x}{3}\right)^x \in \mathcal{O}(x!)$ and pushing even further $\left(\frac{10x}{28}\right)^x \in \mathcal{O}(x!)$, and so we have

$$4 \lim_{g\to\infty} \frac{(2g-2)(2g-3)\dots(g+1)(g)}{(g-1)!(2^{2g})} \leq 4 \lim_{g\to\infty} \frac{(2g-2)(2g-3)\dots(g+1)(g)}{(\frac{10(g-1)}{28})^{g-1}(\sqrt{2}^{4g})}$$

At the nominator we have $(g-1) - \lfloor \frac{g-1}{2} \rfloor$ terms whose value is in the range $[g, \lfloor \frac{3g}{2} \rfloor - 1]$ and $\lfloor \frac{g-1}{2} \rfloor$ terms whose value is in the range $[\lfloor \frac{3g}{2} \rfloor, 2g-2]$. At the denominator, we can multiply $3\sqrt{2}$ to $\lfloor \frac{g-1}{2} \rfloor$ factors and $4\sqrt{2}$ to the remaining ones. Notice that $3\sqrt{2}\frac{10}{28} > 1.5$ and $4\sqrt{2}\frac{10}{28} > 2$. The limit becomes

$$4 \lim_{g\to\infty} \frac{(2g-2)}{\left(\frac{40\sqrt{2}(g-1)}{28}\right)} \frac{(2g-3)}{\left(\frac{40\sqrt{2}(g-1)}{28}\right)} \cdots \frac{\lfloor \frac{3g}{2} \rfloor + 1}{\left(\frac{40\sqrt{2}(g-1)}{28}\right)} \frac{\lfloor \frac{3g}{2} \rfloor}{\left(\frac{40\sqrt{2}(g-1)}{28}\right)} \frac{\lfloor \frac{3g}{2} \rfloor - 1}{\left(\frac{30\sqrt{2}(g-1)}{28}\right)} \cdots \frac{g}{\left(\frac{30\sqrt{2}(g-1)}{28}\right)} \frac{1}{(\sqrt{2})^{4g-k}} = 0$$

where $k = 4\left((g-1) - \lfloor \frac{g-1}{2} \rfloor\right) + 3\lfloor \frac{g-1}{2} \rfloor \geq 4\left(\frac{g-1}{2} + 1\right) + 3\frac{g-1}{2} = \frac{7g+1}{2}$ and hence $\sqrt{2}^{4g-k} = 2^{\frac{g-1}{4}}$. The limit goes to 0 since each factor (except for $1/2^{\frac{g-1}{4}}$ which clearly tends to 0) tends to a finite value smaller than 1. In conclusion, we have that the original limit $\lim_{g\to\infty} \frac{2\sum_{j=0}^{g-1}\left(\frac{(j+g-1)!}{(g-1)!(j)!}\right)}{2^{2g}}$ is 0 and so the thesis follows immediately.

□

Lemma 5. $(n^{2-\frac{1}{62}} \log^2 n) \in \mathcal{O}\left(n^2 \frac{\log\log n}{\log n}\right)$

Proof.

$$\lim_{n\to\infty} \frac{n^{2-\frac{1}{62}} \log^2 n}{n^2 \frac{\log\log n}{\log n}} = \lim_{n\to\infty} \frac{n^2}{n^2} \frac{\log^3 n}{(\log\log n)\sqrt[62]{n}} = 0$$

□