Virginia Vassilevska Williams and Ryan Williams

Last time, we defined linear decision trees (LDTs), showed that APSP can be solved with LDTs in $\tilde{O}(n^{2.5})$ depth, and showed how to get an algorithm for APSP on the Real RAM. Today we'll show how to also get a low-depth LDT for 3-SUM, and show how to get a 3-SUM algorithm on the Real RAM using this LDT. We'll also (optionally) talk about an interesting 3-SUM data structure that was recently developed, refuting a conjecture about the hardness of 3-SUM.

Recall the real-numbered version of k-SUM we want to work with:

Problem: k-SUM
Input: n **real** numbers $r_1, \ldots, r_n$
Output: Indices $(i_1, \ldots, i_k) \in [n]^k$ such that $\sum_j r_{i_j} = 0$.

In particular, the output of k-SUM is an $O(k \log n)$-bit string. Recall the k-SUM conjecture is:

k-SUM Conjecture: For every $k \geq 2$ and $\varepsilon > 0$, k-SUM cannot be solved in $O(n^{\lceil k/2 \rceil - \varepsilon})$ (randomized) time.

We will show this conjecture is false when we replace "time" with "linear decision tree depth".

1 Recalling Linear Decision Trees

Let's quickly recall the definition of LDTs and how they compute.

Definition 1.1 *For fixed integers m and m', a* **linear decision tree** *(LDT) of width ℓ with m inputs and m' bits of output is a* binary *tree T where each inner node of T is associated with some inequality*

$$[\alpha_{i_1} x_{i_1} + ... + \alpha_{i_\ell} x_{i_\ell} \geq t]$$

where $t \in \mathbb{R}$, the x_{i_j} are variables from the set $\{x_1, \ldots, x_m\}$ for some fixed m, and each $\alpha_{i_j} \in \{-1, 0, 1\}$. Each leaf of T is labeled by a string from $\{0,1\}^{m'}$.

T_m computes on an $\vec{x} = (x_1, \ldots, x_m) \in \mathbb{R}^m$ as follows:

- Start at the root of T.
- If the inequality at the current node of T is true of $\vec{x}$, then move to the left child of the current node, else move to the right child.
- Once a leaf is reached, output its string.

Given a function $f : \mathbb{R}^n \to \{0,1\}^m$, the "linear decision complexity" of f is the minimum depth of an LDT that computes f. As before, we will use the following fact:

Fact 1.1 *$SORT$ has an LDT of width 2 and depth $O(n \log n)$.*

2 Low-Depth LDT for 3-SUM

The LDT for APSP (that we saw last time) was from the 1970s. In more recent times, researchers have discovered low-depth linear decision trees for 3-SUM as well, starting with:

Theorem 2.1 ([GP14]) 3-SUM *has an LDT of width 4 and depth* $\tilde{O}(n^{1.5})$. *In general,* k-SUM *has a LDT of width* $(2k-2)$ *and depth* $\tilde{O}(n^{k/2})$.

The most amazing aspect of this theorem is that it took nearly 40 years to prove it, after Fredman's results for APSP! Once you see it, you'll know what I mean. (It builds on our $O(n^2)$-time algorithm for 3-SUM with the two pointers.) Probably the only reason this LDT was not found earlier, was simply because people believed it didn't exist.[1] There was a lower bound:

Theorem 2.2 ([Eri99]) *Every LDT of width 3 for* 3-SUM *has depth at least* $\Omega(n^2)$.

Somehow, when you allow width 4 instead of width 3, all of a sudden 3-SUM gets easier for linear decision trees. Now, since 3-SUM is comparing triples of numbers anyway, people didn't figure that considering width larger than 3 would help much. Erickson's work [Eri99] also shows that k-SUM needs $\Omega(n^{\lceil k/2 \rceil})$ depth for width-k LDTs.

For k-SUM, the state-of-the-art in low-depth LDTs is the following result of Kane, Lovett, and Moran:

Theorem 2.3 ([KLM19]) k-SUM *has a width-*$2k$ *LDT of depth* $O(kn \log^2 n)$.

Therefore, 3-SUM has a width-6 LDT of depth only $O(n \log^2 n)$. In fact, one can determine if there is a subset of *at most* k numbers that sum to zero, using their LDT construction. Therefore this result also implies that even Subset-Sum (the infamous NP-complete problem) has a LDT of depth $\tilde{O}(n^2)$, by setting $k = n$. The fact that Subset-Sum has a poly(n)-depth LDT has actually been known for a long time, thanks to work of Meyer auf der Heide [Mey84] and Meiser [Mei93].

2.1 The LDT

We will use the 3-SUM* variant of 3-SUM (which is equivalent, as seen in Lecture 10):

3-SUM*: Given a set $A \subset \mathbb{R}$, are there $a_i, a_j, a_k \in A$ such that $a_i = a_j + a_k$?

Without loss of generality, let's assume all numbers in A are distinct. (We can check if a 3-SUM instance contains a 3sum solution with two repeats of the same number, in $\tilde{O}(n)$ time.)

Recall the $O(n^2)$ time algorithm we gave for 3-SUM:

1. Sort A in $O(n \log n)$ time.
2. For each a in A,
   Make two pointers on the A: p_1 at the beginning of L, and p_2 at the end.
3. Repeat until the pointers reach each other:
   Let b be the current number at p_1 and c be the number at p_2.
   If $a = b$, move p_1 to the right *(we want a distinct triple of numbers)*
   If $a = c$, move p_2 to the left *(same reason)*
   If $a + b + c = 0$ then return (a, b, c).
   If $a + b + c > 0$, then move p_2 to the left *(to get a smaller 3-sum, we have to decrease c)*
   If $a + b + c < 0$, then move p_1 to the right *(to get a larger 3-sum, we have to increase b)*
4. Return "no solution".

[1] This is an important point to reflect on, especially given all these fine-grained hypotheses and conjectures we've seen!

For each $a \in A$, step 3 takes $O(n)$ time to find the other two numbers in the 3sum solution.

Exercise: Argue that the above algorithm can be implemented by a width-3 LDT of depth $O(n^2)$.

We want to speed-up the loop in step 3. The starting idea is to *partition* the sorted A into consecutive groups of numbers $A_1, \ldots, A_{n/d}$, with $O(d)$ numbers in each group. For two groups A_j and A_k, we want to *quickly* check if there are $b \in A_j$ and $c \in A_k$ such that $a = b + c$. If we can do that in time (well, depth) t, then we'd only have to repeat the loop in step 3 for n/d times, and get $O(t \cdot n/d)$ time. We will show how to do some sorting tricks to get that depth t down to $O(\log d)$.

Here is our new proposed algorithm. It is underspecified; we have to fill in more details to complete it!

1. Sort A in $O(n \log n)$ time. Partition A into groups $A_1, \ldots, A_{n/d}$ with $O(d)$ numbers in each group.
1.1 For all $i \in [n/d]$,[2] define the list $D_i := \{a_j - a_k \mid a_j, a_k \in A_i\}$.
Sort each D_i in $\tilde{O}(d^2)$ time/depth. Merge all D_i lists into a single sorted list L in $\tilde{O}(nd)$ time/depth.
1.2 For all $j, k \in [n/d]$, define the list $S_{j,k} := \{a_j + a_k \mid a_j \in A_j, a_k \in A_k\}$.
2. For each $a_i \in A$,
Make two pointers on the *groups* of A: p_1 at A_1, p_2 at $A_{n/d}$.
3. Repeat until the pointers reach each other:
Let A_j be the group that p_1 points to, and A_k be the group p_2 points to.
If $(a_i \in S_{j,k})$ **then** return (i, j, k) as a solution.
Otherwise, $a_i \neq a_j + a_k$, for all $a_j \in A_j$ and $a_k \in A_k$.
Set $a_{j,max}$ to be the largest number in A_j, and $a_{k,min}$ to be the smallest in A_k.
If $(a_i < a_{k,min} + a_{j,max})$, then move p_2 to the left *(to get a smaller 3-sum, decrease the "larger" group)*
else move p_1 to the right. *(to get a larger 3-sum, we have to increase the smaller group)*
4. Return "no solution".

Observe that the sorting in step 1.1 takes $\tilde{O}(n/d \cdot d^2) \leq \tilde{O}(nd)$ depth with a width-4 LDT (by Fact 1.1), and the number of iterations in step 3 is at most n/d.

Exercise: Convince yourself that the above algorithm correctly solves 3-SUM*!

However, step 1.2 looks suspect! There are n^2/d^2 lists $S_{j,k}$, and each of these lists is $\Theta(d^2)$ long, if all of the sums $a_j + a_k$ are distinct. How do we prepare the lists $S_{j,k}$ in $\ll n^2$ depth, and how do we implement the **bold-faced** check of $(a_i \in S_{j,k})$ efficiently? Well, we did define these lists D_i which don't seem to be doing anything...

The key insight to answering these questions is the following.

Key Insight: Any two elements of any $S_{j,k}$ can be compared using the sorted order on L.

Take any $a_j, a'_j \in A_j$ and $a_k, a'_k \in A_k$, so that $a_j + a_k$, $a'_j + a'_k$ are from $S_{j,k}$. Recall Fredman's trick from the last lecture:

$$a_j + a_k \leq a'_j + a'_k \iff a_j - a'_j \leq a'_k - a_k.$$

Note that $a_j - a'_j \in D_j$ and $a'_k - a_k \in D_k$. Therefore, both of these differences appear in the list L, which we sorted in step 1.1. Thus any two elements of $S_{j,k}$ can be compared using L. Therefore, after step 1.1, we can already infer the sorted order of every $S_{j,k}$.

[2] Remember $[K] := \{1, \ldots, K\}$.

Using this insight, we claim:

Claim: Each run of step 3 can be implemented using only $O((n/d) \cdot \log d)$ depth. In particular, the check $(a_i \in S_{j,k})$ can be done in only $O(\log d)$ depth.

Since it takes $\tilde{O}(nd)$ depth to implement steps 1 and 1.1, and we run step 3 for up to n times, the claim would imply that the total depth is (omitting O-tilde factors)

$$nd + (n^2/d) \cdot \log(d).$$

Setting $d = n^{1/2}$, we would therefore obtain an LDT of depth $\tilde{O}(n^{1.5})$.

Let's see how to prove the claim. Note for each j, k, the size of $S_{j,k}$ is $|S_{j,k}| \leq O(d^2)$.

> **Exercise:** Show that, assuming we know the sorted order of each $S_{j,k}$, we can determine $(a_i \in S_{j,k})$ for any i, j, k, with a width-3 LDT of depth only $O(\log d)$.

Assuming the exercise, the whole of step 3 only needs $O(n/d \cdot \log d)$ additional depth: we have to do only $O(n/d)$ iterations of the loop, and one extra comparison in each loop to move the pointers.

Getting a Real RAM Algorithm. You can use the LDT above to get a 3-SUM algorithm on the Real RAM that runs in $O(n^2/\log^c n)$ time for some $c > 0$. The idea is very similar to the APSP algorithm: we reduce the problem on n numbers to very small sub-instances of a numbers, for some $a = \log^b n$ where $b > 0$, and store in memory an LDT for 3-SUM on a numbers so that we can save a bit over the trivial $O(a^2)$ time algorithm. The best known algorithm in this line of work runs in $O(n^2 \text{poly}(\log\log n)/(\log n)^2)$ time, by Chan [Cha20].

3 An Interesting "Static" Data Structure for 3-SUM (Optional)

Over the next few lectures, we will discuss data structure lower bounds in fine-grained complexity. So this is a good place to give a recently-found data structure for 3-SUM, which refuted a conjecture in the area. The data structure was inspired by ideas from cryptography. We consider the data structure problem, 3-SUMINDEXING:

> **Problem:** 3-SUM INDEXING
> **Input:** A list L of n integers.
> **Task:** Preprocess L so that, given any integer a, we can quickly determine a pair $(b, c) \in L^2$ such that $a = b + c$, if one exists.

In such a problem, there are two phases. In the *preprocessing phase*, an algorithm is run directly on the list L, and some data structure of size $S(n)$ is prepared.[3] In the *query phase*, we receive "queries" which are integers a, and for each query we want to answer in some $T(n)$ time if there are $b, c \in L$ such that $a = b + c$. Our data structure should work quickly no matter what integer a is given to us. We would like both phases to be as efficient as possible: both space $S(n)$ used in the preprocessing phase, and the time $T(n)$ used in the query phase. These data structures are often called *static* in that the initial input does not change, to contrast with *dynamic* data structures where the underlying input can be modified over time.

There are two simple ways to solve 3-SUM INDEXING. The first way would be to simply store all $O(n^2)$ possible pairs $(b, c) \in L^2$ along with their sum. Then on any query, we can lookup the answer in $O(\log n)$ time. This solution has a horrible space usage for preprocessing, but fast *query time*. Another way would be to store nothing but L, and

[3] We generally don't care much about the running time of the preprocessing phase, but ideally it should be small, like polynomial time. The point is that the preprocessing would be run only once.

solve the resulting 2-SUM instance (using the two-pointer algorithm) in $O(n \log n)$ time. This has no preprocessing at all, but linear query time. Can we get sub-linear query time $T(n)$ with sub-quadratic space $S(n)$ for preprocessing? It was conjectured that the answer is no:

Conjecture 3.1 ([GKLP17]) *For all $\delta > 0$, every data structure for* 3-SUM INDEXING *with $O(n^{1-\delta})$ query time must take at least $\Omega(n^2)$ space for preprocessing.*

Assuming this conjecture, some other lower bounds were proved. This conjecture was recently refuted:

Theorem 3.1 ([GGH+20]) *There is a data structure for* 3-SUM INDEXING *that can be prepared in $\tilde{O}(n^2)$ time with $O(n^{1.9})$ space for preprocessing and $O(n^{0.3})$ query time.*

That is, we can store only $n^{1.9}$ bits of information about all $O(n^2)$ possible pairwise sums from a set of size n, and still be able to determine if there is a pair that sums to a given target in only $O(n^{0.3})$ time!

In general, for every $k \geq 3$ and every $\delta \in (0, k-2)$, the k-SUM INDEXING problem (which asks for a $(k-1)$-tuple of numbers that sum up to a given target b) can be solved with $O(n^{k-1-\delta})$ preprocessing space and $\tilde{O}(n^{3\delta})$ query time.

To illustrate the key points in this theorem, we will make one simplification: *we will assume that all numbers in our* 3-SUM INDEXING *instance are distinct integers drawn from $\{1, \ldots, n^2\}$.* This is actually not such a "huge" simplification, since we could hash the numbers modulo $O(\log n)$ random primes p that are close to n^2 (as we did in our 3-SUM algorithm in Lecture 9) and preserve the correct answer with good probability. But it is a simplification.

Next, we reduce our indexing problem to a classic cryptography problem: *function inversion*. Let $L = \{a_1, \ldots, a_n\}$ be a given list of n (distinct) integers in $\{1, \ldots, n^2\}$.

Let $\rho : [n^2] \to [n]^2$ be a bijection that takes every integer in $[n^2]$ to a unique pair $(i, j) \in [n]^2$, and let $\rho_1, \rho_2 : [n^2] \to [n]$ be such that $(\rho_1(x), \rho_2(x)) = \rho(x)$. Define a function $f : [n^2] \to [n^2]$ as follows:

$$f(x) := \text{Parse } x \text{ into } i = \rho_1(x),\ j = \rho_2(x), \text{ and output } a_i + a_j.^4$$

Observe that the 3-SUM INDEXING problem is *precisely* the problem of building a data structure for inverting the function f! A query consists of a number b, and we want to find (i, j) such that $a_i + a_j = b$, if they exist: this is exactly the problem "given a b, find an x such that $f(x) = b$". Function inversion is the one of most basic problems in cryptography, and it has been studied for decades.

It turns out that non-trivial (interesting) data structures for function inversion exist! The following very generic theorem shows how to do it:

Theorem 3.2 ([FN00]) *For any integer $X \geq 1$, any $f : [X] \to [X]$, and any S and T such that $S^3T \geq X^3$, there is a data structure using space $\tilde{O}(S)$ which can invert f at any desired point $b \in X$, which uses at most $\tilde{O}(T)$ time and $\tilde{O}(T)$ evaluations of f.*

Assuming Theorem 3.2, we can solve 3-SUM INDEXING by preparing a data structure for the f defined above, with $X = n^2$ and $S = n^{1.9}$. Solving for T, we see that T can be $n^6/n^{5.7} \leq n^{0.3}$, and we are done!

That leaves the question of *how* to prove Theorem 3.2. The full proof of Theorem 3.2 would be too much to cover (even optionally) so we will illustrate a simple case.

By far the nicest case is when $f : [X] \to [X]$ is a bijection. In this case, we can show that there is a data structure using $\tilde{O}(S)$ space that can invert f in $\tilde{O}(T)$ time, for all $S \cdot T \leq X$.

Define a graph G_f with nodes $1, \ldots, X$ and the X directed edges $(x, f(x))$ for all $x \in [X]$. Since f is a bijection, every node has indegree 1 and outdegree 1, so G_f is a collection of disjoint cycles. For every cycle of length at least X/S, conceptually break it up into disjoint paths such that all but possibly one of the paths is of length X/S. From these, we form a list of disjoint paths $P_1, \ldots, P_k$. Note there are at most $O(S)$ such paths in total, so $k \leq O(S)$. For

[4] Technically, to keep the co-domain $[n^2]$, we should only output $a_i + a_j$ if $a_i + a_j \leq n^2$, and otherwise output a "non-answer" such as 1.

each path P_i, suppose it starts at a node s_i and ends at a node t_i. Our data structure stores the numbers at nodes s_i and t_i, and a new "back edge" (t_i, s_i); this takes $O(S \log X)$ bits of space. Note that G_f augmented with the edges (t_i, s_i) now has the property that every node is in a cycle of length at most $O(X/S)$.

Now suppose we are given a $y \in [X]$, and want to know the unique x such that $f(x) = y$ (remember, f is a bijection). This is equivalent to finding the node x in G_f such that (x, y) is an edge. To do this, we go *forwards* in the graph, rather than backwards: imagine we have a pointer to a node v in G_f; initially, $v := y$. We evaluate f on v, conceptually taking the edge $(v, f(v))$. If $f(v) = t_i$ for some $i = 1, \ldots, k$, we set $v := s_i$ (taking the edge (t_i, s_i)). We repeat this, until we take the edge (v, y), in which case we output v as the x such that $f(x) = y$.

Why does this work? Recall that G_f is a collection of disjoint cycles; by repeatedly computing f on the current value, we are traveling along a cycle and will eventually reach the node we started from. Moreover, in the above procedure, we will not visit more than $O(X/S)$ nodes, because if our input y happens to be on a P_i of length X/S, after at most $O(X/S)$ nodes we will take the "back edge" (t_i, s_i), and end up at x such that (x, y) is an edge after $O(X/S)$ steps. (If y was not on a path of length X/S, then it must be in a cycle of length less than X/S, so we will reach x in $O(X/S)$ evaluations of f.)

To sum up, the above procedure uses $T \leq O(X/S)$ time and evaluations of f, and inverts f on all inputs.

Note that our data structure described above *crucially* requires that f is a bijection; otherwise the graph G_f could have lots of other weird structure in it. This makes the full proof of Theorem 3.2 quite complicated.

4 Open Problems

The following is one of the major open problems in data structures: find a *static* data structure task which has poly(n) possible queries on n-bit instances, and where every data structure using at most $n^{1+\varepsilon}$ space to preprocess the queries will require at least $(\log n)^{1+\varepsilon}$ time to answer the queries, for some $\varepsilon > 0$. (Really, the open problem is to find any problem that needs both super-linear space and super-logarithmic query time.)

A less ambitious open problem would be to study other weaker forms of the 3-SUM INDEXING Conjecture mentioned that haven't yet been refuted, and try to prove more interesting consequences of them (or refute them!). Two other weaker conjectures are mentioned in [GGH+20].

References

[Cha20] Timothy M. Chan. More logarithmic-factor speedups for 3sum, (median, +)-convolution, and some geometric 3sum-hard problems. *ACM Trans. Algorithms*, 16(1):7:1–7:23, 2020.

[Eri99] Jeff Erickson. Bounds for linear satisfiability problems. *Chic. J. Theor. Comput. Sci.*, 1999, 1999.

[FN00] Amos Fiat and Moni Naor. Rigorous time/space trade-offs for inverting functions. *SIAM Journal on Computing*, 29(3):790–803, 2000.

[GGH+20] Alexander Golovnev, Siyao Guo, Thibaut Horel, Sunoo Park, and Vinod Vaikuntanathan. Data structures meet cryptography: 3sum with preprocessing. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 294–307. ACM, 2020.

[GKLP17] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In Faith Ellen, Antonina Kolokolova, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures - 15th International Symposium, WADS*, volume 10389 of *Lecture Notes in Computer Science*, pages 421–436. Springer, 2017.

[GP14] Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *J. ACM*, 65(4):22:1–22:25, 2018. Conference version in FOCS'14.

[KLM19] Daniel M. Kane, Shachar Lovett, and Shay Moran. Near-optimal linear decision trees for k-sum and related problems. *J. ACM*, 66(3):16:1–16:18, 2019.

[Mei93] Stefan Meiser. Point location in arrangements of hyperplanes. *Inf. Comput.*, 106(2):286–303, 1993.

[Mey84] Friedhelm Meyer auf der Heide. A polynomial linear search algorithm for the n-dimensional knapsack problem. *J. ACM*, 31(3):668–676, 1984.