6.S078 Fine-Grained Algorithms and Complexity **MIT**
Lecture 12: Linear Decision Trees and APSP Over the Reals **Wednesday October 14, 2020**

In this lecture, we will discuss an interesting computational model, and how APSP with real-valued weights (and eventually, 3-SUM with real numbers) can be solved more efficiently in this model. We will consider functions of the form

$$f : \mathbb{R}^m \to \{0,1\}^{m'}$$

where m and m' are positive integers. When appropriately redefined slightly, the APSP problem (over reals) and 3-SUM (over reals) fit the above form. For example, we can define

> **Problem:** APSP (a.k.a. Min-Plus Matrix Multiplication)
> **Input:** Two $n \times n$ matrices A and B over $\mathbb{R}$
> **Output:** For all $i, j \in [n]$, output a $k \in [n]$ such that $A[i,k] + B[k,j]$ is minimized.[1]

Recall we said (and more-or-less proved) that APSP and Min-Plus Matrix Multiplication have equivalent running times. Note that the output of APSP as defined above is $O(n^2 \log n)$ bits: $O(\log n)$ bits for every $i, j \in [n]$. The APSP Conjecture is:

> **APSP Conjecture:** There is no $\varepsilon > 0$ such that APSP can be solved in $O(n^{3-\varepsilon})$ time.[2]

We saw several problems (such as Negative Weight Triangle) which are equivalent to APSP.

In this class, we will redefine k-SUM to be:

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

So, we can think of the output of k-SUM as being 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.

Although we formulated this conjecture for the integer version of k-SUM, it also makes perfect sense to study for the real version as well.

So far, the only computational model we have seen for solving problems on real-valued inputs is the Real RAM, which lets us do additions, subtractions, and comparisons with contents of real-valued registers, and word operations in "normal" word registers of $\Theta(\log n)$ bits.[3]

---

[3]Here we recall the main features of the Real RAM model, as seen in Lecture 7. This model is an extension of the *Word RAM* model. In the Word RAM, information is stored in "word registers" that hold $\log(n)$ bits, and you can do all the normal word operations on them (XOR two words into a third word, AND two words into a third word, etc) in constant time. In the Real RAM, there are also "real-valued registers", each of which can hold an arbitrary real number. We are allowed to add two real-registers and put the result in another real register in $O(1)$ time, and one can compare if a real A is at most a real B, and put the (Boolean) outcome of that comparison into a word register, in $O(1)$ time. (The Real RAM is often also called the "addition-comparison model".) We will assume that the input to APSP is given to us in n^2 real-valued registers, and the input to 3-SUM is given in n real-valued registers.

In this lecture, we will study a *non-uniform* model that captures what the Real RAM can do, and show **there is a surprisingly *faster* algorithm for APSP in this model**. Later, we will also see such an algorithm for 3-SUM. These fast non-uniform algorithms are what led to the first speed-ups in the Real RAM for both of these problems. Studying these algorithms more closely may help us understand the extent to which the APSP and 3-SUM Conjectures are actually true.

0.1 A Brief Aside About Non-Uniform Models

In a *non-uniform* computational model, the program we get to run can vary, based on the length of the input. We are allowed a program P_1 that we get to run on inputs of length 1, another program P_2 that we run on inputs of length 2, and in general a program P_n that we run on inputs of length n, for all n. A canonical example of a non-uniform complexity class is P/poly: this is the class of functions $f : \{0,1\}^\star \to \{0,1\}$ such that there is a polynomial $p(n)$ and an infinite family of Boolean logical circuits $\{C_n\}$, where for all n the function $f_n : \{0,1\}^n \to \{0,1\}$ (the function f restricted to n-bit inputs) is computed by C_n, and the number of gates in C_n is at most $p(n)$. It is known that $P \subset \mathsf{P}/\mathrm{poly}$, but the containment is proper! Note that there is no computational requirement on how long it might take to "construct" a circuit C_n in the family.

Exercise: Prove that P/poly contains an undecidable language!

Nevertheless, it is a major open problem to prove that $\mathsf{NP} \not\subset \mathsf{P}/\mathrm{poly}$, or to even prove that $\mathsf{NTIME}[2^n] \not\subset \mathsf{P}/\mathrm{poly}$. (By the end of the semester, you will see an unexpected connection between fine-grained complexity and such problems!)

1 The Model: Linear Decision Trees (LDT)

We start with the definition.

Definition 1.1 *A* **linear decision tree** *(LDT) of width ℓ 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 $\{0,1\}^{m'}$.[4]

We generally want the width ℓ to be very small, e.g., a constant. That way, in the Real RAM, each inequality at each node of the LDT could be "computed" in $O(1)$ time, provided that these $x_{i,j}$ and t are sitting in real-valued registers.[5]

How LDTs Compute. Next, we describe how computation works in an LDT T. By the definition, each T works on a fixed number of variables 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.

[4]Note: the use of 0 as a coefficient implies that we allow any inequality of any number of variables between 1 and ℓ.

[5]Note: if the $\alpha_{i,j}$ were arbitrary reals, instead of being from $\{-1, 1\}$, then the functions computed at the inner nodes of the LDT are linear threshold functions (LTFs), which are also studied in neural networks and in circuit complexity. It's not clear whether extending the domain of the coefficients gives any new power, but it probably would!

- 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.

(Note that the very special case where each inequality is of the form $x_i \geq 1$ for Boolean variables $x_i \in \{0, 1\}$, corresponds to simply *decision trees*. This special case is extremely well-studied, and their powers and limitations are well-known.)

LDTs correspond to a non-uniform model of computation, because for every number of variables m, the LDT could be very different. And there is *no* computational cost on the *construction* of such an LDT. It could be that the leaves and inequalities involved might take a long time to determine computationally, even knowing what the other inequalities in the LDT are.

Complexity of LDTs. We measure the "running time" of an LDT by its *depth*, the maximum length of a path from the root to a leaf, because that's the maximum possible number of inequalities thatwould need to be computed in order to determine the answer for a given input, and each inequality would take $O(\ell)$ time in the Real RAM.

So, 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.

To sanity-check that this definition of complexity makes sense, let's consider a well-known function, sorting. We think of the problem of sorting real numbers as a function $SORT : \mathbb{R}^n \to \{0, 1\}^{O(n \log n)}$, where the output is a bit string encoding a permutation of $\{1, \ldots, n\}$ (encoding how the given list of reals should be sorted).

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

In particular, all inequalities in this LDT can be made to have the form $x_i - x_j \leq 0$, to compare two variables x_i and x_j. As a very simple example, here is an LDT for sorting a list of two numbers:

If $[x_1 - x_2 \leq 0]$ then output $(1\ 2)$ else $(2\ 1)$.

That is, if $x_1 \leq x_2$ then we output the identity permutation on 2 elements, otherwise we output the inverse permutation.

We'll critically use Fact 1.1 in our APSP and 3-SUM algorithms later.

Exercise: Prove Fact 1.1.

2 A Low-Depth LDT for APSP

The first surprising LDT we'll cover is for APSP.

Theorem 2.1 (Fredman [Fre75]) *APSP has an LDT of width 4 and depth $O(n^{2.5} \log^{1/2} n)$.*

This is a truly-subcubic algorithm for APSP! However, there is a catch: it heavily exploits the non-uniformity of LDTs. Recall in our earlier APSP algorithm, we showed that

$$(\min, +) \text{ matrix multiplication on } n \times d \times n \text{ matrices in } T(n, d) \text{ time}$$

$$\Longrightarrow (\min,+) \text{ matrix multiplication on } n \times n \times n \text{ matrices in } O(n/d \cdot (T(n,d) + n^2)) \text{ time,}$$

by simply splitting the $n \times n \times n$ matrix multiplication into n/d separate $n \times d \times n$ matrix multiplications, after which we take the component-wise minimium of n/d matrices where each matrix is $n \times n$. This reduction also works when we replace "time" with "LDT depth": this reduction only computes minima of numbers via comparisons, which can be done with an LDT. We will use this reduction with the following claim:

Claim 2.1 *For $d \leq n$, $(\min,+)$ matrix multiplication on $n \times d$ and $d \times n$ matrices has an LDT of width 4 and $O(nd^2 \log n)$ depth.*

For $d \ll n$, this depth is surprisingly low: we can infer an output of n^2 real numbers with only about nd^2 additions and comparisons! (On some level, this looks impossible: we are inferring what the n^2-size output of a function is, by doing only about $nd^2 \ll n^2$ operations. It turns out that there is some "redundancy" in that output which can be exploited.)

Assuming the Claim, and using our reduction, we can get an LDT for min-plus multiplication of $n \times n$ matrices, of depth $O(n/d \cdot (nd^2 \log n + n^2)) \leq O(n^2 d \log n + n^3/d)$. Setting $d = \sqrt{n/\log n}$, we obtain an $O(n^{2.5} \log^{1/2} n)$-depth LDT, and Fredman's theorem!

Let's now prove the claim. We are given $A \in \mathbb{R}^{n \times d}$ and $B \in \mathbb{R}^{d \times n}$, and want to determine the $n \times n$ matrix:

$$C(i,j) = \min_{k=1,\ldots,d} (A[i,k] + B[k,j]).$$

Let's restate our task as:

For every $i, j \in [n]$, find $k^\star \in [d]$ such that for all $k \in [d]$, $A[i,k^\star] + B[k^\star,j] \leq A[i,k] + B[k,j]$.

These $k^\star$ will tell us the minimum $A[i,k] + B[k,j]$. The key to our LDT is "Fredman's trick", which is the "deep" fact that computing $\min_{k=1,\ldots,d}(A[i,k] + B[k,j])$ is equivalent to finding $k^\star$ such that for all k, $A[i,k^\star] - A[i,k] \leq B[k,j] - B[k^\star,j]$. Here's a high-level description of the LDT:

0. Intialize L to be an empty list.
1. For all $i \in [n]$, and all pairs $(k,k') \in [d]^2$,
   Append $a_{i,k,k'} := A[i,k'] - A[i,k]$ to the list L. Append $b_{i,k,k'} := B[k,i] - B[k',i]$ to L.
2. Sort L. *By Fact 1.1, this can be done with a width-4 LDT in $O(n \cdot d^2 \log n)$ depth.*
3. For every $i, j \in [n]^2$, consider the sublist of L on just the $a_{i,k,k'}$ and $b_{j,k,k'}$ entries (there are $2d^2$ of them). *The ordering of these $a_{i,k,k'}$ and $b_{j,k,k'}$* **determines** *the minimum $k^\star$ such that $A[i,k] + B[k,j]$. Thus, our LDT can now output such $k^\star$, for every i, j.*

Note: no additions or comparisons are needed in step 3. *The matrix C can be immediately output after step 2 is done, because all of its entries are actually determined by the ordering of the elements of L.*

Exercise: Verify the italicized claims above.

3 Using the LDT to Get a Real RAM Algorithm

The width-4 LDT for APSP is quite a funny "algorithm" and it's certainly exploiting non-uniformity: sure, the matrix C is "determined" once we know the ordering on all the $a_{i,k,k'}$ and $b_{j,k,k'}$, but we didn't really calculate how much *work* it would actually take to figure out each of the entries of C from the sorted order. So the next question is, how can we get a Real RAM algorithm from this?

The only step we don't know how to do in the Real RAM (uniform) model is step 3: We don't know how to *quickly* determine the output matrix C from the sorted L of size $O(nd^2)$. However, we can give a Real RAM algorithm that removes some log-factors from the $O(n^3)$ running time. It's possible that there exists a much better way to use LDTs to solve APSP (beyond the $n^3/2^{\Omega(\sqrt{\log n})}$-time algorithm that we sketched in a previous lecture).

Theorem 3.1 (Fredman [Fre75]) *APSP can be computed in $O(n^3/\log^{1/7} n)$ time on the Real RAM.*

The $\log^{1/7} n$ factor is not optimized; it is just meant to be simple and illustrative. Fredman's original algorithm got a better improvement than $\log^{1/7} n$, but it was still below $\log n$.

Proof. Recall it suffices to compute $n \times n$ min-plus matrix multiplication in $O(n^3/\log^{1/7} n)$ time. Our idea is to build a small LDT that quickly computes min-plus matrix multiplication on small matrices, and reduce our $n \times n$ matrix mult to calls on these small matrices. In $2^{O(a^{2.5}\log^{1/2} a)} \cdot \mathrm{poly}(a)$ time, we can build a width-4 LDT of depth $O(n^{2.5} \log n)$ for min-plus matrix multiplication on $a \times a$ matrices, by mimicking the above LDT at every step.

Exercise: Verify that a width-4 LDT for $a \times a$ matrices could be constructed in $2^{O(a^{2.5}\log^{1/2} a)} \cdot a^k$ time, for some constant $k \geq 1$. You may assume that the LDT for sorting a list of ℓ elements can be constructed in $2^{O(\ell \log \ell)}$ time, in case your proof of Fact 1.1 didn't do that.

Note that this LDT could be stored in "normal" word registers: we do not need real numbers to store it, as the coefficients are $\{0, 1, -1\}$ and the threshold value t is 0. We can store each node of this LDT in a separate word. Then, given two $a \times a$ matrices with real-valued entries, we can compute the min-plus matrix multiplication of $a \times a$ matrices in $O(a^{2.5} \log^{1/2} a)$ time on the Real RAM, by simulating the LDT on our matrices (starting from the root of the LDT) and computing the four additions/comparisons at each node in $O(1)$ time.

Setting $a := (\log n)^{1/3}$, observe that it takes only $n^{o(1)}$ time to build the LDT (assuming you did the above exercise). Then, given an $n \times a$ matrix A' and an $a \times n$ matrix B' that we wish to min-plus multiply, we can break A' and B' into n/a blocks:

$$A' = \begin{bmatrix} A'_1 \\ \vdots \\ A'_{n/a} \end{bmatrix}, \; B' = [B'_1, \ldots, B'_{n/a}],$$

where each block A'_i and B'_j is an $a \times a$ matrix. Using our LDT of depth $O(a^{2.5} \log^{1/2} a)$ on all pairs of blocks (one from A' and one from B'), we obtain an algorithm running in $O((a^{2.5} \log^{1/2} a) \cdot n^2/a^2) \leq O(n^2 \cdot a^{1/2} \log^{1/2} a)$ time for computing $n \times a \times n$ min-plus matrix multiplication.

Finally, to compute $n \times n$ min-plus matrix multiplication, we reduce the problem to computing n/a matrix multiplications of $n \times a \times n$, then apply our algorithm for $n \times a \times n$ multiplication. The final running time is $O(n/a \cdot n^2 \cdot a^{1/2} \log^{1/2} a) \leq O(n^3/\log^{1/7} n)$. □

The best-known APSP algorithm which *avoids* using fast matrix multiplication (and builds significantly on the above) is due to Chan [Cha10], which runs in $O(n^3(\log\log n)^3/(\log n)^2)$ time (and yes, it also uses "Fredman's trick").

4 A Low-Depth LDT for 3-SUM

The LDT from the previous section was from the 1970s. In more recent times, researchers have discovered low-depth linear decision trees for 3-SUM as well, starting with:

Theorem 4.1 ([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 $\tilde{O}(n^{k/2})$.*

We'll discuss how to prove this and other results in the next lecture. 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.[6] There was a lower bound:

Theorem 4.2 ([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 4.3 ([KLM19]) *k-SUM has a width-$2k$ LDT of depth $O(kn\log^2 n)$.*

In fact, one can determine if there is a subset of *at most k* numbers that sum to zero, with their LDT construction. Therefore this result also implies that 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].

References

[Cha10] Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.*, 39(5):2075–2089, 2010.

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

[Fre75] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5(1):83–89, 1976. Conference version in FOCS'75.

[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.

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

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