COSC 240, Spring 2022: Computational Complexity

Term paper on

Randomization and Logspace

Ankita Sarkar

Abstract

We survey derandomization results for space complexity, with a focus on logarithmic space. First, we discuss derandomization in the realm of decision problems, via the landmark 1999 result by Saks and Zhou which implies that $\mathsf{BPL} \subseteq \mathsf{L}^{3/2}$ and, in further generality, establishes the corresponding statement for any space bound $S(n) \geq \log n$. Then, we move to the realm of search problems, and discuss a different "partial" notion of derandomization called *reproducibility*: via a 2019 result by Grossman and Liu, we see that problems in $\mathsf{search\text{-}RL}$ admit randomized algorithms which can *reproduce* the same answer in successive executions, (almost) irrespective of the internal randomness, and using $O(\log n)$ space.

---

1 Introduction

In this survey, we study the notion of derandomization, i.e. of simulating probabilistic computations using "less random" computations, while preserving (or almost preserving) the desired computational bounds. We focus on space bounds – in particular, on logarithmic space-bounded computation. For such computation, we look at two notions of derandomization: one for decision problems, and another for search problems. In decision problems, the computation returns either `ACCEPT` or `REJECT` on each input, i.e. it decides a language; but in search problems, an answer that is a function of the input must be computed. For example, the decision version of VERTEX-COVER asks *whether or not* the graph G has a vertex cover of size at most k; but the search version asks for such a vertex cover to be *produced*.

Derandomizing space-bounded computation for decision problems. A natural notion of derandomization of decision problems is to replace randomized algorithms with deterministic algorithms while preserving (or almost preserving) complexity guarantees. Under this notion, we study a result by Saks and Zhou [SZ99] which, for $S(n) \geq \log n$, derandomizes bounded-error probabilistic (BP) $S(n)$-space bounded computations to obtain deterministic $O(S(n)^{3/2})$-space bounded computations. In particular, this means that $\mathsf{BPL} \subseteq L^{3/2}$. Saks and Zhou achieved this result by viewing the computation of a probabilistic Turing machine (PTM) as the powering of a matrix: the $(i,j)^{th}$ index of this matrix contains the probability that, from configuration i, the computation will reach configuration j in one step. Thus, to simulate the PTM computation deterministically, one can compute the $(\texttt{INITIAL}, \texttt{ACCEPT})$ entry of a large enough power of this matrix. In fact, since the goal is to simulate *bounded-error* computation, it is only necessary to distinguish between probabilities $\frac{2}{3}$ and $\frac{1}{3}$, and a close estimate of the desired probability will preserve this gap; which means that a good approximation of the powered matrix will suffice. Saks and Zhou thus arrive at their result by approximating the repeated squaring of a matrix, within the desired space bound.

Reproducibility for search problems. It is interesting to consider, for search problems, how the internal randomness affects the answer produced. In particular, we wonder if it is possible to derandomize an algorithm in the following sense: for different executions of the algorithm, each with fresh internal randomness, we want the answer produced to be the same (almost) every time. This notion is called *reproducibility*. For the complexity class search-RL, of search problems that can be solved with probability $1/\text{poly}(n)$ via randomized logspace algorithms, Grossman and Liu [GL19] proved a reproducibility result in 2019. They established that problems in search-RL admit randomized logspace algorithms that, given an instance, outputs the same answer for a large fraction of the internal random choices. To achieve this, Grossman and Liu introduce the notion of *influential bits* of an algorithm, i.e. those (hopefully few) bits of the internal randomness that affect the answer with high probability. They first establish that, to prove logspace reproducibility in the above sense, it suffices to exhibit a randomized logspace algorithm with $O(\log n)$ influential bits. Then, they provide such an algorithm for a search-RL-complete problem.

Organization of this survey. In Section 2, we establish preliminary definitions. In Section 3, we discuss the derandomization result by Saks and Zhou [SZ99] that implies $\mathsf{BPL} \subseteq L^{3/2}$. In Section 4, we formalize the notion of reproducibility, and study the logspace reproducibility of search-RL as shown by Grossman and Liu [GL19].

2 Preliminaries

Throughout, we fix an arbitrary alphabet Σ and refer to languages over this alphabet. We also use the acronyms DTM and PTM, respectively, for deterministic and probabilistic Turing machines. The number n always denotes the input size. We will use T for Turing machines instead of M, because we want to use M for matrices.

Recall that

Definition 1 (DSPACE)**.** *For a language $L \subseteq \Sigma^*$ and a function $S(n)$, $L \in \mathsf{DSPACE}(S(n))$ if there exists a DTM T that runs in $O(S(n))$ space and $\forall x \in \Sigma^*$, $T(x) = L(x)$.*

We can similarly define the bounded-error probabilistic (BP) version of the above:

Definition 2 (BPSPACE)**.** *For a language $L \subseteq \Sigma^*$ and a function $S(n)$, $L \in \mathsf{BPSPACE}(S(n))$ if there exists a PTM T that runs in $O(S(n))$ space making some random choices r, and $\forall x \in \Sigma^*$,*

$$x \notin L \implies \Pr_r[T(x,r) = 1] \leq \frac{1}{3}, \text{ and}$$
$$x \in L \implies \Pr_r[T(x,r) = 1] \geq \frac{2}{3}.$$

We also define $\mathsf{BPL} := \mathsf{BPSPACE}(\log n)$.

In Section 3, we will crucially use the following definitions about matrices over reals.

Definition 3 (substochastic matrix)**.** *A square matrix M is said to be* substochastic *if all its entries are non-negative and each row sums to at most* 1*.*

Definition 4 (L_1-norm)**.** *For a vector $x \in \mathbb{R}^d$, the L_1-*norm of x*, denoted $\|x\|$, is $\sum_{i=1}^m x_i$. For a $d \times d$ matrix over $\mathbb{R}$, the L_1-*norm of M*, denoted $\|M\|$, is $\max_{i=1}^m \|M(i,\cdot)\|$, i.e. the maximum of the L_1-norms of its rows.*

Definition 5 (truncation)**.** *For a $z \in \mathbb{N}$ and $t \in \mathbb{N}$, $\lfloor z \rfloor_t$ is the* truncation of z to t bits*, obtained by taking the first t bits of the binary expression for z. For a matrix M, $t \in \mathbb{N}$, the* truncation of M to t bits*, $\lfloor M \rfloor_t$, is obtained by truncating each entry to t bits, i.e. replacing each entry of the form $M(u,v)$ with $\lfloor M(u,v) \rfloor_t$.*

In Section 4, we will study randomized algorithms for search problems. To motivate the formal definition of a search problem, consider the familiar decision problem VERTEX-COVER $= \{\langle G, k\rangle \mid G$ has a vertex cover of size $\leq k\}$. The corresponding search problem would be to *find* the vertex cover instead of only asserting its existence, i.e. search-VERTEX-COVER$= \{(\langle G, k\rangle, W) \mid W$ is a vertex cover in G and $|W| \leq k\}$. Moreover, for an input $\langle G, k\rangle$, it is useful to consider the set of solutions for it, i.e. the set search-VERTEX-COVER$(\langle G, k\rangle) = \{W \mid (\langle G, k\rangle, W) \in$ search-VERTEX-COVER$\}$; this set is empty iff $\langle G, k\rangle \notin$ VERTEX-COVER.

In the general definition below, x is an input, like $\langle G, k\rangle$; y is a desired output, like a vertex cover W of size at most k; R corresponds to search-VERTEX-COVER, L_R corresponds to VERTEX-COVER, and $R(x)$ corresponds to search-VERTEX-COVER$(\langle G, k\rangle)$.

Definition 6 (search problem, [GL19, Definition 2.1])**.** *A search problem is a relation R consisting of pairs (x, y). We define $L_R = \{x \mid \exists y : (x, y) \in R\}$, and $R(x) = \{y \mid (x, y) \in R\}$.*

Recall the class RL of decision problems that admit randomized logspace algorithms with perfect soundness.

Definition 7. [1] *For a language $L \subseteq \Sigma^*$, $L \in \mathsf{RL}$ if there exists a PTM T that runs in $O(\log n)$ space making some random choices r, and $\forall x \in \Sigma^*$,*

$$x \notin L \implies \Pr_r[T(x,r) = 1] = 0, \text{ and}$$
$$x \in L \implies \Pr_r[T(x,r) = 1] \geq \frac{1}{\text{poly}(|x|)}.$$

search-RL is precisely the search version of the above, i.e. the class of search problems that admit randomized logspace algorithms with perfect soundness, in the sense that if no solution exists, a search-RL algorithm always detects that, whereas if a solution exists, the algorithm finds *one* such solution with probability at least $\frac{1}{\text{poly}(n)}$.

Definition 8 (search-RL, rewording of Definition 2.5, Grossman and Liu [GL19])**.** *A search problem R is in* search-RL *if there is a randomized logarithmic space transducer T such that $\forall x \in L_R$, $\Pr_r[T(x,r) \in R(x)] \geq \frac{1}{\text{poly}(|x|)}$.*

To prove results about search-RL, we naturally require complete problems for the class. Consider the following problem Short-Walk Find Path where, given a graph G and a source s, and the fact that most k-length walks starting from s end at a vertex t, the task is to search for an s-t path of length at most k.

Definition 9 (Short-Walk Find Path [GL19, Definition 2.6])**.** *Let R be the search problem whose valid inputs are $x = (G, s, t, 1^k)$ where G is a directed graph, s and t are two vertices of G, and a random walk of length k from s reaches t with probability at least $1 - 1/|x|$. On such an x, a valid output is a path of length up to k from s to t.*

Note that k is represented in unary so that a logspace algorithm has to be logspace in terms of k as well. Grossman and Liu show [GL19, Appendix B] that Short-Walk Find Path is search-RL-complete. We do not reproduce the proof here, but we appeal to the following intuition: recall that stcon $= \{\langle G, s, t\rangle \mid$ there is a path in G from s to $t\}$ is NL-complete, and Short-Walk Find Path can be viewed as a "randomized search" version of stcon.

3 Derandomizing BPL

In this section, we discuss the following result by Saks and Zhou [SZ99]. This result makes an assumption that our space bounds are in terms of *proper complexity functions*. This assumption is true [SZ99] for the functions we come across in the day-to-day – logarithmic, polynomial, exponential, and the like – so we will not belabor this assumption.

Theorem 3.1 ([SZ99, Theorem 1.1])**.** *Let $S(n)$ be a proper complexity function such that $S(n) \geq \log n$. Then*

$$\mathsf{BPSPACE}(S(n)) \subseteq \mathsf{DSPACE}(S(n)^{3/2})$$

Saks and Zhou reinterpret their derandomization task as the task of approximating *repeated matrix squaring*: for a $d \times d$ matrix M, and parameters r, a, i, j, the task is to approximate $M^{2^r}(i,j)$ *with accuracy* 2^{-a}, which means to produce an answer in $[M^{2^r}(i,j) - 2^{-a}, M^{2^r}(i,j) + 2^{-a}]$. So let us first see why this reinterpretation is valid.

[1] Via error-reduction techniques, the $\frac{1}{\text{poly}(n)}$ in this definition can be replaced with $\frac{1}{2}$, which is often taken as the definition of RL. We choose the $\frac{1}{\text{poly}(n)}$ definition for consistency with Grossman and Liu's definition of search-RL.

3.1 Reduction to repeated matrix squaring

Fix a language $L \in \mathsf{BPSPACE}(S(n))$, and let T be the corresponding PTM. Also fix an input x, $|x| = n$. Let the number of configurations in the resulting computation of T be $d(n)$; note that $d(n)$ is at most $2^{\Theta(S(n))}$. WLOG, assume that T has unique `INITIAL`, `ACCEPT`, and `REJECT` states. Consider the following matrix M, where the rows and columns are indexed by the configurations: the $(i,j)^{th}$ entry $M(i,j)$ is the probability that, from configuration i, the computation in T goes to configuration j in one step. Then, for any number $k \geq 1$, $M^k(i,j)$ is the probability that, from configuration i, the computation goes to configuration j in k steps. So

$$\Pr_r[T(x,r) = 1] = M^{d(n)}(\texttt{INITIAL}, \texttt{ACCEPT})\,.$$

Now consider the following equivalent view of the computation in T: instead of the computation halting at `ACCEPT` and `REJECT` states, once the computation reaches either of those states, it loops at that state forever. We reinterpret $T(x,r) = 1$ as when we loop forever at `ACCEPT`, and $T(x,r) = 0$ as when we loop forever at `REJECT`. Under this view, we can extend the matrix interpretation above to say that

$$\forall k \geq d(n),\ M^k(\texttt{INITIAL}, \texttt{ACCEPT}) = M^{d(n)}(\texttt{INITIAL}, \texttt{ACCEPT}) = \Pr_r[T(x,r) = 1]\,.$$

So if we choose $r(n) = \lceil \log d(n) \rceil$, and we can efficiently compute $M^{2^{r(n)}}(\texttt{INITIAL}, \texttt{ACCEPT})$, then we have the following simple derandomization procedure: if $M^{2^{r(n)}}(\texttt{INITIAL}, \texttt{ACCEPT}) \geq \frac{2}{3}$, then accept, else reject.

In reality, we are unable to efficiently compute $M^{2^{r(n)}}(\texttt{INITIAL}, \texttt{ACCEPT})$ *exactly*, but let us see how *approximating* it with sufficient accuracy is enough. Suppose, for a parameter $a \geq 1$, we could efficiently obtain an answer $A \in [M^{2^{r(n)}}(\texttt{INITIAL}, \texttt{ACCEPT}) - 2^{-a}, M^{2^{r(n)}}(\texttt{INITIAL}, \texttt{ACCEPT}) + 2^{-a}]$. If $x \in L$, $A \geq \frac{2}{3} - 2^{-a}$, and if $x \notin L$, $A \leq \frac{1}{3} + 2^{-a}$. By choosing $a = 3$, this suggests the following simple derandomization procedure: if $A \geq \frac{13}{24}$, then accept, else reject. Intuitively, we used the large-ish *gap* between the soundness and completeness errors in the definition of $\mathsf{BPSPACE}$ to ensure that, for a large enough accuracy parameter a, the approximate answer preserves the gap.

Thus, Saks and Zhou are able to obtain their result by providing an algorithm to approximate repeated matrix squaring. They do so for *substochastic* matrices (see Definition 3), and the above configuration matrices of a PTM computation are indeed substochastic. They ensure that, for a $d \times d$ substochastic matrix M, approximating an entry of M^{2^r} uses $O(r^{1/2} \cdot \max\{r, \log d\})$ space. So when M is as above, with $d = d(n) = 2^{\Theta(S(n))}$, and $r = r(n) = O(\log d(n)) = O(S(n))$, the final algorithm uses space

$$O(r^{1/2} \cdot \max\{r, \log d\}) = O(O(S(n))^{1/2} \cdot \max\{O(S(n)), \Theta(S(n))\} = O(S(n)^{3/2})\,.$$

We will now see their algorithm for approximating the repeated squaring of a substochastic matrix. Formally, this problem is stated as follows:

> Approximate Repeated Matrix Squaring for Substochastic Matrices (ARMS-Substochastic)
>
> **Input.** A $d \times d$ substochastic matrix M, entry parameters $i, j \in [d]$, power parameter $r \in \mathbb{N}$, accuracy parameter $a \geq 1$.
>
> **Output.** A number in $[M^{2^r}(i,j) - 2^{-a}, M^{2^r}(i,j) + 2^{-a}]$.

The main result of this section is now reduced to the following result:

Theorem 3.2 (Rewording of Theorem 3.1, Saks and Zhou [SZ99])**.** *There is a deterministic algorithm for* ARMS-Substochastic *such that, when* $r, a \in O(\log d)$*, the algorithm has space complexity* $O(\log^{3/2} d)$*.*

Saks and Zhou produce a randomized algorithm for ARMS-Substochastic with the property that, if it uses $O(r^{1/2} \cdot \max\{r, \log d\})$ random bits and $O(r^{1/2} \cdot \max\{r, \log d\})$ computation space, then it can be derandomized to give an $O(r^{1/2} \cdot \max\{r, \log d\})$-space deterministic algorithm. This property is obtained via the notion of *off-line randomization*, which we will not explain in this survey. We will focus on why their algorithm is correct and of the desired space complexity.

3.2 The Saks and Zhou [SZ99] randomized algorithm for ARMS-Substochastic

By replacing r with the smallest perfect square that is at least r, the algorithm assumes WLOG that r is a perfect square. Now let

$$\Lambda(M) := M^{2^{\sqrt{r}}}.$$

Then composing Λ with itself some s times gives

$$\Lambda^{(s)}(M) = \underbrace{\Lambda \circ \cdots \circ \Lambda}_{s \text{ times}}(M) = M^{2^{s\sqrt{r}}}$$

i.e. $\Lambda^{(\sqrt{r})}(M) = M^{2^r}$.

Overall framework and use of random bits. Via an earlier result due to Nisan [Nis90], Saks and Zhou obtain a procedure Nisan that approximates $\Lambda(M)$ to sufficient accuracy, using $O(r)$ random bits. Then they use Nisan as a subroutine to compute $\Lambda^{(\sqrt{r})}(M)$, which takes $\sqrt{r}$ calls to Nisan, using additional $O(\max\{r, \log d\})$ random bits for processing steps after each call. Thus, per call to Nisan, they use $O(r) + O(\max\{r, \log d\}) = O(\max\{r, \log d\})$ random bits, for a total of $O(r^{1/2} \cdot O(\max\{r, \log d\})$ random bits.

The algorithm works as follows. In this presentation, we have used some asymptotic notation for simplicity; where precise constants are important, the relevant choices can be found in the paper [SZ99].

procedure Saks-Zhou(M, r, a, i, j)
  $M \leftarrow \lfloor M \rfloor_{O(\sqrt{r})}$ ▷ truncate entries to the first $O(\sqrt{r})$ bits
  $s \leftarrow 0$ ▷ counts calls to Nisan
  while $s < \sqrt{r}$ **do**
    $M^\Lambda \leftarrow$Nisan(M) ▷ Approximate $\Lambda(M)$, uses $O(r)$ random bits
    $s \leftarrow s + 1$ ▷ update counter
    Pick $\delta \in_R [-2^{-O(\max\{r, \log d\})}, 2^{-O(\max\{r, \log d\})})$ ▷ Pick a small number δ using extra random bits
    $\forall u, v \in [d], M^\delta(u, v) \leftarrow \max\{M^\Lambda(u, v) - \delta, 0\}$ ▷ decrease entries of M^Λ by the small random number, but keep them non-negative
    $M \leftarrow \lfloor M^\delta \rfloor_{O(\sqrt{r})}$ ▷ truncate entries to the first $O(\sqrt{r})$ bits, a la Definition 5
  return $M(i, j)$

We will use M_i to refer to M at the end of the i^{th} iteration of the algorithm, and δ_i to be the random δ picked during the i^{th} iteration. So $M_0 = \lfloor M \rfloor_{O(\sqrt{r})}$, and $M_{\sqrt{r}}(i,j)$ is the output. By extension, we will say that $M_i^\Lambda = \textsc{Nisan}(M_{i-1})$, and M_i^δ is obtained when we reduce the entries of M_i^Λ by δ_i.

Space complexity. Note that, given M_i^Λ, computing M_i^δ and M_{i+1} each take $O(\max\{r, \log d\})$ space, since there are d^2 entries in the matrices, each at most $O(\sqrt{r})$ bits long. Furthermore, Nisan is guaranteed [Nis90, SZ99] to require $O(\max\{r, \log d\})$ space. Thus, over $\sqrt{r}$ iterations, the desired space complexity is attained.

Correctness. Consider the following hypothetical procedure: we perform the same steps as the Saks-Zhou procedure, but instead of using recursive calls to Nisan, we compute Λ exactly. Let the matrices formed by this procedure be as follows: $N_0 = \lfloor M \rfloor_{O(\sqrt{r})}$, and for $i \geq 1$, N_i is the matrix formed at the end of the i^{th} iteration, $N_i^\Lambda = \Lambda(N_{i-1})$, and N_i^δ is the matrix formed when we decrease the entries of N_i^Λ by δ_i.

There are two key ideas in the proof by Saks and Zhou. The first is that, with high probability over their random choices, the N_i's and M_i's are the same; the second is that, for any choices of δ_i's, $N_{\sqrt{r}} = M^{2^r}$. For this exposition, we do not want to focus on the specifics of the random choices, as they are derandomized later anyway. So we will detail the following part of the proof: we will assume that good random choices are made, and show how this implies that the M_i's equal the N_i's.

We formalize this in Lemma 3.3 below. We will use the L_1-norm, as per Definition 4. The first property assumed in Lemma 3.3 follows from good choices of δ, while the second one follows from the good behavior of Nisan.

Lemma 3.3 (Distilled version of Lemma 5.2, Saks and Zhou [SZ99])**.** *Suppose the following properties hold (via good random choices and the correctness of* Nisan*):*

1. $\forall i \in [\sqrt{r}]$, $\forall u, v \in [d]$, $N_i^\delta(u,v) - N_i(u,v) \notin [-2^{-O(\max\{r,\log d\})}, 2^{-O(\max\{r,\log d\})})$, *and*
2. $\forall i \in [\sqrt{r}]$, $\|M_i^\Lambda - N_i^\Lambda\| \leq 2^{-O(\max\{r,\log d\})}$.

Then $\forall i \in [\sqrt{r}], M_i = N_i$.

Proof. We induct on i. The base case is simply that $N_0 = \lfloor M \rfloor_{O(\sqrt{r})} = M_0$. Now inductively assume that $M_{i-1} = N_{i-1}$. We can check from the definition of M_i^δ and N_i^δ that $\|M_i^\delta - N_i^\delta\| \leq \|M_i^\Lambda - N_i^\Lambda\|$, which by Property 2 is at most $2^{-O(\max\{r,\log d\})}$. By Definition 4, then

$$\forall u, v \in [d], |M_i^\delta(u,v) - N_i^\delta(u,v)| \leq 2^{-O(\max\{r,\log d\})} \tag{1}$$

Now we crucially apply Property 1 along with the fact that $N_i = \lfloor N_i^\delta \rfloor_{O(\sqrt{r})}$. For any $u, v \in [d]$, it must be that either $N_i^\Lambda(u,v) \in [0, 2^{-O(\max\{r,\log d\})} - 2^{-O(\sqrt{r})})$, or $N_i^\delta(u,v) - N_i(u,v) \in [2^{-O(\max\{r,\log d\})}, 2^{-O(\sqrt{r})} - 2^{-O(\max\{r,\log d\})})$. Combined with (1), this means that

$$\forall u, v \in [d], M_i^\delta(u,v) - N_i^\delta(u,v) \in [0, 2^{-O(\sqrt{r})}) .$$

So $\forall u, v \in [d], \lfloor M_i^\delta \rfloor_{O(\sqrt{r})} = \lfloor N_i^\delta \rfloor_{O(\sqrt{r})}$, i.e. $M_i = N_i$. ∎

Summary of details behind Lemma 3.3. We have not explained above how Nisan actually works, or why it is actually a good approximation for Λ. One crucial detail is that the random bits

passed to NISAN must be *pseudorandom w.r.t. N_i's*; i.e., under these random choices, NISAN must approximate each $N_i^\Lambda = \Lambda(N_i)$ with some sufficient accuracy K, where K is a function of a and $\max\{r, \log d\}$. Once this relative notion of pseudorandomness is established, Property 2 will hold. This same K is used to pick δ in a way that Property 1 holds; so indeed δ depends on a as well; but, in the simple presentation above, we have used the bounded-error property of BPSPACE to pretend that a is, say, 3.

4 Reproducibility of search-RL

We first discuss the notion of *reproducibility* for search problems (Definition 6) as formalized by Grossman and Liu [GL19] in 2019. For a search problem R, to formalize the notion of "reproducing the same answer", Grossman and Liu view an algorithm with reproducibility as a composition of two algorithms: on input x, algorithm A produces (with high probability) a "fixer" t_x; the other algorithm B, using both input and fixer i.e. (x, t_x), outputs a fixed answer $y \in R(x)$ (again, with high probability). This is formalized as follows.

Definition 10 (reproducible, [GL19, Definition 3.1])**.** *A search problem R has* logspace reproducible solutions *if there exist randomized logspace algorithms A and B such that*

- *On input x, with probability at least $\frac{2}{3}$, A outputs a string t_x of length $O(\log n)$ such that ...*
- *... there exists some y satisfying $(x, y) \in R$ such that with probability at least $\frac{2}{3}$, B on input (x, t_x) outputs y.*

Grossman and Liu also present the following alternate view of reproducibility: the existence of a randomized logspace algorithm C that, on input x, returns two copies of the same output y with high probability. They show that this notion is equivalent [GL19, Lemma 3.2] to Definition 10.

Now, we study their proof [GL19, Section 3] that problems in search-RL (Definition 8) admit reproducible solutions. The first key idea is the relationship between reproducibility and *influential bits*, i.e those bits that *truly* affect the output. Once this relationship is established, they design an algorithm for the search-RL-complete problem SHORT-WALK FIND PATH (Definition 9) with a number of influential bits small enough to imply reproducibility.

4.1 Reproducibility as a bound on the number of influential bits

Suppose, out of the many random bits that an algorithm uses, there are $O(\log n)$ bits such that, with high probability, the output only depends on them. Since we are allowed $O(\log n)$ space, the algorithm could then remember these bits, and reuse them to reproduce the answer. This is formalized as follows:

Definition 11 (influential bits, [GL19, Definition 3.3])**.** *For a polytime computable function $k(n)$, a randomized logspace search algorithm A has $k(n)$* influential bits *if for all valid inputs x, with probability at least $\frac{1}{2}$ over random strings $r_1 \in \{0,1\}^{k(n)}$, there exists y, a valid output for x such that $\Pr_{r_2}[A(x, r_1, r_2) = y] \geq \frac{2}{3}$, where r_2 is the remaining randomness used by A after using r_1.*

Lemma 4.1 ([GL19, Lemma 3.4])**.** *If a search problem R admits a randomized logspace algorithm with $O(\log n)$ influential random bits, then it has logspace reproducible solutions.*

Proof sketch. Let C be a randomized logspace algorithm for R with $O(\log n)$ influential bits. Via error-reduction, we can replace the $\frac{1}{\text{poly}(n)}$ in the definition of search-RL (Definition 8) with $\frac{1}{\exp(n)}$. So we assume that if there is a valid output for an input, then C finds it with probability at least $1 - 1/\exp(n)$.

Now we construct the algorithms A and B required by the definition of reproducibility (Definition 10). On input x, the algorithm A generates $O(\log n)$ random bits, t_x, and tests whether $\Pr_{r_2}[C(x, t_x, r_2) = y] \geq 1 - 1/n^2$. Using some repetition, it finds a suitable t_x which it can output as the required "fixer". The algorithm B then simulates C on (x, t_x).

More formally, the algorithm A works as follows. We assume that an output is always of length at most $m(n) := \text{poly}(n)$, and that C uses $r(n)$ random bits. Then, we compare $C(x, t_x, r_2)$ for many choices of r_2, as a way of testing whether $\Pr_{r_2}[C(x, t_x, r_2) = y] \geq 1 - 1/n^2$. There is, however, a caveat: $C(x, t_x, r_2)$ is the *output*, so it is allowed to have $\omega(\log n)$ bits, i.e. we cannot store all of $C(x, t_x, r_2)$ in $O(\log n)$ space to compare it to the outputs from subsequent r_2's. So instead, we use freshly random repetitions for each i^{th} bit of the output, and execute the subroutine Is-Reproducible, which tests whether the i^{th} bit is the same for many choices of r_2; finally, we want Is-Reproducible to return `true` for each i.

```
1:  procedure A(x)
2:      flag ← true                                  ▷ Tracks whether a suitable t_x is found
3:      count ← 0                                    ▷ Tracks number of attempts at finding t_x
4:      while flag = false and count < O(log n) do
5:          Pick t_x ∈_R {0,1}^{O(log n)}            ▷ A possible fixer, which will now be tested
6:          for i = 1 to m(|x|) do ▷ Tests if the i^th bit of output is reproduced using t_x
7:              flag ←Is-Reproducible(x, t_x, i)     ▷ subroutine as shown below, to test
    whether i^th bit is reproduced
8:              if flag = false then                 ▷ bad fixer, break and try another one
9:                  break
10:         count ← count + 1
11:     return t_x
```

where the subroutine Is-Reproducible is as follows.

```
1:  procedure Is-Reproducible(x, t_x, i ∈ [m(|x|)])
2:      Pick r_2 ∈_R {0,1}^{r(n)−O(log n)}
3:      y ← C(x, t_x, r_2)(i) ▷ i^th bit of output using r_2. Subsequent loop will compare this to
    freshly random repetitions
4:      test_flag ← true
5:      count′ ← 1                                   ▷ Tracks iterations of testing loop
6:      while count′ < Θ(n^2) do
7:          Pick r_2 ∈_R {0,1}^{r(n)−O(log n)}       ▷ Fresh randomness for repeating
8:          test_flag ← test_flag ∧ (C(x, t_x, r_2)(i) = y)   ▷ only remains true if C outputs
    the same i^th bit for each r_2
9:          count′ ← count′ + 1
10:     return test_flag
```

Once we have established algorithm A, algorithm B is simple, as follows.

```
1: procedure B(x, t_x)
2:     Pick r_2 ∈_R {0,1}^{r(n)−O(log n)}
3:     return C(x, t_x, r_2)
```

In algorithm A, because we check the output one bit at a time, we can reuse $O(\log n)$ space for each bit. For a choice of t_x, we can check that, if $\Pr_{r_2}[C(x, t_x, r_2)]$ is at least $1 - 1/n^2$, then `flag` will become true with high probability. We can also check that a suitable t_x will be found in $O(\log n)$ attempts.[2] These facts ensure that algorithm A, with high probability, outputs t_x such that $\Pr_{r_2}[C(x, t_x, r_2)] \geq 1 - 1/n^2$, which means that B also outputs the same answer on (x, t_x) with high probability over r_2. ■

Now, to show the reproducibility of problems in search-RL, it suffices to produce algorithms for them with $O(\log n)$ influential bits.

4.2 Algorithms for search-RL problems with $O(\log n)$ influential bits

Grossman and Liu exhibit an algorithm [GL19, Algorithm 1] for the search-RL-complete problem Short-Walk Find Path, which has $O(\log n)$ influential bits. This implies that all problems in search-RL admit such algorithms, and hence

Theorem 4.2 ([GL19, Theorem 3.1])**.** *Every problem in* search-RL *has a randomized logspace algorithm that has only $O(\log n)$ influential bits.*

The algorithm. Intuitively, Grossman and Liu's algorithm is a randomized version of the standard NL algorithm for stcon. It works as follows.

Given an input graph G, vertices u, v, and a number l, let $p_l(u, v)$ [GL19, Definition 2.8] be the probability that a random walk of length l starting from u passes through v. For now, assume that we are able to compute $p_l(u, v)$ for free, given G, u, v, l. Then consider the following algorithm on the input (G, s, t, k): start from s, and go over the out-neighbors of s to check if, for some out-neighbor v, $p_{k-1}(v, t)$ is suitably large (above a threshold we will fix later); if yes, then move to v, decrement k, and recurse. The random walk thus performed should go through t with high probability, and to return a path, we can output each v when we go to it. We can reuse $O(\log n)$ space at each level of the recursion, so that the overall algorithm is logspace.

In reality, we cannot compute the probabilities $p_l(u, v)$ exactly. But Grossman and Liu show [GL19, Lemma 2.9] that we can obtain, with high probability, an estimate $\mu_l(u, v) \in [p_l(u, v) - \varepsilon, p_l(u, v) + \varepsilon]$ for $\varepsilon = O(1/k^5 n^5)$. So, instead of checking whether p_l's are suitably large, our algorithm will recurse depending on whether μ_l's are suitably large. Next, we need to fix what "suitably large" is.

Ideally, we want that the random bits used to pick μ_l's should not be influential, so a suitable threshold should be of the form $1/2 - c$ where $|1/2 - c - p_l(u, v)| > \varepsilon$ for all vertices u, v: this way, since $|p_l(u, v) - \mu_l(u, v)| \leq \varepsilon$, our choice of whether or not to recurse will only depend on the true p_l's, despite the fact that the algorithm checks the μ_l's. This suggests, since $\varepsilon = O(1/k^5 n^5)$, that c should be $\geq O(1/k^4 n^4)$. Since the random bits used to get μ_l's are not influential, we can then use $O(\log n)$ random bits to generate c: Grossman and Liu do exactly that, and pick c u.a.r. from the set $\{\frac{1}{k^4 n^4}, \frac{2}{k^4 n^4}, \ldots, \frac{k^2 n^2}{k^4 n^4}\}$. The number of random bits thus used is $\log(k^2 n^2) = 2\log(kn) \leq 2\log(n^2) = 4\log n$.

[2] Technically, we could make more attempts, or even keep trying until a good t_x is found, because there are no time constraints. But since $O(\log n)$ attempts suffice with high probability, that is what we do.

This gives the following algorithm [GL19, Algorithm 1]. We assume that Est-Prob is the algorithm [GL19, Lemma 2.9] that, on input $u, v \in V(G)$ and a number l, produces an estimate $\mu_l(u, v)$ of $p_l(u, v)$.

```
1: procedure Grossman-Liu(G, s, t, 1^k)
2:     curr ← s                                          ▷ Tracks current vertex
3:     Pick c ∈_R {1/(k^4 n^4), 2/(k^4 n^4), ..., k^2 n^2/(k^4 n^4)}   ▷ uses O(log n) bits
4:     l ← k                                             ▷ Tracks remaining length of path
5:     while l > 0 do
6:         Print curr on the output tape ▷ so the path found is output by the end of the algorithm
7:         for each out-neighbor v of curr do
8:             μ ←Est-Prob(l, curr, v)                   ▷ estimate μ_l(u, v) for p_l(u, v)
9:             if μ ≥ 1/2 − c then                       ▷ if p_l(curr, v) is large enough
10:                curr ← v                              ▷ move to next vertex in path
11:                l ← l − 1                             ▷ decrement length of remaining path
```

Analysis. Grossman and Liu show that

Lemma 4.3 ([GL19, Lemma 3.5])**.** *The procedure* Grossman-Liu

1. *runs in $O(\log n)$ space,*
2. *has $O(\log kn)$ influential bits, and*
3. *with high probability, outputs a path from s to t*
4. *in expected polynomial time.*[3]

Proof sketch. We summarize the key arguments towards the proof, which follow from our exposition preceding the algorithm. We omit the precise calculations from this survey.

1. **Space complexity.** Storing c and the name of `curr` requires $O(\log n)$ bits, since $k = O(n)$. μ_l's are obtained [GL19, Lemma 2.9] in $O(\log n)$ space as well, and this space can be reused for each $l \in [k]$.
2. **Influential bits.** For each $u, v \in V(G)$ and $l \in [k]$, we ensured via choice of c that with high probability, $|\mu_l(u, v) - p_l(u, v)| = o(|1/2 - c - p_l(u, v)|)$. Hence, the condition in Line 9 of Grossman-Liu effectively only depends on the true p_l values, i.e. the random bits used to estimate μ_l's are not influential. So the only random bits that influence the output are the $O(\log kn)$ bits required to choose c.
3. **Correctness.** Est-Prob succeeds with high probability [GL19, Lemma 2.9], so via a union bound argument, all k executions of it succeed with high probability.
4. **Running time.** The key idea here is that if $p_l(\texttt{curr}, t) \geq 1/2 - c$, then for some out-neighbor v of `curr`, $p_{l-1}(v, t) \geq 1/2 - c$; so with high probability, `curr` will change in each iteration.

■

[3] This is a bonus guarantee. Theorem 4.2 does not require this.

References

[GL19] Ofer Grossman and Yang P. Liu. Reproducibility and pseudo-determinism in log-space. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 606–620, 2019.

[Nis90] Noam Nisan. Pseudorandom generators for space-bounded computation. In *Proc. 22nd ACM Symposium on Theory of Computing, 1990*, pages 204–212, 1990.

[SZ99] Michael Saks and Shiyu Zhou. $BP_HSPACE(S) \subseteq DSPACE(S^{3/2})$. *Journal of Computer and System Sciences*, 58(2):376–403, 1999.