6.5420 Randomness and Computation February 12, 2026

Lecture 4

Lecturer: Ronitt Rubinfeld *Scribe: Yina Wang*

1 Overview

This lecture finishes the discussion of Moser's Algorithm from the previous lecture and introduces the theory of random walks on graphs (stationary distributions, hitting/cover times, and properties).

2 Moser's Algorithm Cont.

Recall that the input to the algorithm is a random string

$$R \in \{0,1\}^{|R|}.$$

The algorithm reads bits of R in blocks of size l whenever it performs a call to Fix(), and uses the first n bits for the initial assignment. We encode the entire execution trace as follows.

2.1 Encoding Scheme

Let $b \in \{0,1\}^m$ be the bit string where

$$b_i = \begin{cases} 1 & \text{if Fix}(C_i) \text{ is called at the top level,} \\ 0 & \text{otherwise.} \end{cases}$$

For each i with $b_i = 1$, we encode the full recursive call tree rooted at Fix(C_i). For the top level, each $b_i = 1$ is associated with an extra bit, which is 1 if this top-level call has any recursive children. Each node in such a tree corresponds to a recursive call to Fix(C_j) and stores:

1. The index of the clause relative to its parent in the adjacency list (requires $\log_2(d+1)$ bits).
2. A child bit (1 if this node has a recursive child call, 0 otherwise).
3. A sibling bit (1 if it has a recursive sibling call, 0 otherwise).

Thus each node uses $\log_2(d+1)+2$ bits in the encoding. In the end, the encoding E consists of:

1. The string b (length m),
2. All recursive trees (in DFS order, with an extra bit for top-level calls to Fix(), indicating the existence of recursive child calls),
3. The final variable assignment (length n),
4. All unused bits of R.

In the succeeding sections, we denote by S the total number of calls to Fix() (including recursive calls).

2.2 Reconstructing R from the Encoding E

We now explain why the encoding is injective by describing how to reconstruct the original random string R from the encoding E. We do this in two passes.

In the first pass,

1. The first m bits give the string b.
2. For each position i with $b_i = 1$, we reconstruct the corresponding recursive call tree. Specifically, we read the DFS encoding of the tree, using the information contained in the child and sibling bits to construct the edges in the trace of computation. Because the encoding is written in DFS order, we can reconstruct the full tree structure deterministically.
3. The next n bits give the final variable assignment.
4. The remaining bits are exactly the unused suffix of R.

At this point, we know the full execution trace (including the exact recursive structure), the final assignment of variables, and the unused suffix of R.

In the second pass, we reconstruct the used bits of R by reversing the execution of the algorithm. Recall that each call to Fix(C) consumed exactly l fresh random bits to rerandomize the variables in clause C.

We process the calls in reverse DFS order. Specifically, starting from the final variable assignment, we consider the last call to Fix(C). Immediately before that call, clause C was unsatisfied; there is a unique assignment of the variables in this clause that would make all of the literals (and thus the entire clause) evaluate to false. Looking at the final assignment for the variables involved in this clause, we recover the l random bits that were used in that call. We then restore the variables to their pre-call values and continue backward.

Since each call to Fix() rerandomizes only the variables of its clause, and since an unsatisfied clause determines a unique falsifying assignment, the reconstruction is deterministic.

After reversing all calls, we recover all lS bits used in recursive calls, and the initial n bits used for the first assignment. Appending the unused suffix gives the full original string R.

Thus the encoding is injective.

Example 1. *Consider the run with*

$$R = 000001100\ 110\ 000\ 101\ 110\ 11001.$$

Last lecture, we argued that the proper encoding for this example is

$$E = 1010\ 1\ 0110\ 0000\ 0\ 101101110\ 11001$$

We first parse that $b = 1010$, there are thus two recursive trees corresponding to the first and third top-level calls, the final variable assignment, and the unused suffix 11001*. In the first pass, we reconstruct the trace. Specifically, from $b = 1010$, we know:*

- Fix(1) *was called. The next bit after b is 1, so there exists a subtree, and we process the next four bits 0110. The second neighbor clause was fixed, and it has a child but no sibling; since the second neighbor clause is 2 from the adjacency list, we know that* Fix(2) *was called recursively, and that the next four bits are its child call. The next four bits are 0000, so node 2's first neighbor clause was fixed, and it has neither a child nor sibling call. So the branch of the trace ends here with* Fix(1).
- Fix(2) *was not called at top level,*
- Fix(3) *was called, but the next bit is 0, so we know there is no subtree.*

- Fix(4) *was not called.*

In summary, we reconstructed the recursive structure:

$$\text{Fix}(1) \to \text{Fix}(2) \to \text{Fix}(1), \quad \text{Fix}(3).$$

Thus there were four total calls to Fix.
In the second pass, we now reverse the calls.

Last call: Fix(3).
The final variable assignment has that $(x_4, x_8, x_9) = (T, T, F)$, so fixing the variables must have required consuming these bits from R:

$$TTF.$$

Second-to-last call: Fix(1). *Immediately before the previous call, (x_4, x_8, x_9) were all false, so we can rewind before the call by assigning $(x_4, x_8, x_9) = (F, F, F)$. Now, the current variable assignment has that $(x_1, x_2, x_3) = (T, F, T)$, so fixing the clause must have required consuming these bits from R:*

$$TFT.$$

Third-to-last call: Fix(2). *Immediately before the previous call, (x_1, x_2, x_3) were all false, so we can rewind before the call by assigning $(x_1, x_2, x_3) = (F, F, F)$. Now, the current variable assignment has that $(x_1, x_2, x_5) = (F, F, F)$, so fixing the variables must have required consuming these bits from R:*

$$FFF.$$

First call: Fix(1). *Immediately before the previous call, $(x_1, x_2, x_5) = (T, T, F)$, so we can rewind before the call by assigning $(x_1, x_2, x_5) = (T, T, F)$. Again, the current assignment now has that $(x_1, x_2, x_3) = (T, T, F)$, and the bits used were:*

$$TTF.$$

The key observation is that an unsatisfied clause carries information. Immediately before a call to Fix(C)*, the clause C must have been false, which means its l variables were in the* unique *assignment that falsifies C. After the call, those same variables were rerandomized using l fresh bits of R. Thus, when we run the execution backward, we know exactly what the variables must have been before the call, and we also know what they are after the call. So we can determine what l bits the rerandomization consumed from R by looking at the current assignment of the variables, and we can rewind before the fix call by choosing the assignment of variables that makes the clause evaluate to False.*

Finally, reading off the initial assignment (by reversing the first assignment step), we recover the first nine bits as 000001100. Putting everything together, we reconstruct

$$R = 000001100\ 110\ 000\ 101\ 110\ 11001.$$

Thus the encoding uniquely determines the original random string.

2.3 Length of the Encoding

Let W denote the total number of random bits actually used by the algorithm. Since $W = n + lS$, we have that the number of unused bits is

$$|R| - W = |R| - n - lS.$$

The total encoding length is therefore

$$|E| = m + S(\log_2(d+1) + 2) + n + (|R| - n - lS),$$

where the first term is the number of bits used to describe b, the second term is the number of bits used in the trace, the third term is the number of bits in the output assignment, and the last term is the number of unused bits. Canceling n and moving $|R|$, we obtain

$$|E| - |R| = m + S(\log_2(d+1) + 2 - l).$$

2.4 Using the Lovász Local Lemma Assumption

We assumed that

$$2^{2.1} \cdot p \cdot (d+1) \le 1, \quad p = 2^{-l}.$$

Taking logarithms gives

$$\log_2(d+1) \le l - 2.1.$$

Therefore,

$$\log_2(d+1) + 2 - l \le (l - 2.1) + 2 - l = -0.1.$$

Plugging in,

$$|E| - |R| \le m - 0.1S.$$

So, when S is sufficiently long, we have that the size of the trace is much smaller than the size of R, and we have successfully described an encoding scheme. Specifically, if $S \ge 10(m+b)$, then

$$|E| - |R| \le m - 0.1 \cdot 10(m+b) = m - (m+b) = -b.$$

Thus

$$|E| \le |R| - b.$$

In this case, the encoding compresses R by b bits.

Note that if f denotes the deterministic mapping from R to E, the number of strings compressible by b bits is at most $2^{|R|-b}$, out of $2^{|R|}$ total strings in the domain. This is because f is injective, i.e., we can recover R from E.

Therefore, the probability that a given string is compressible is bounded by

$$\Pr\left[|E| \le |R| - u\right] \le \frac{2^{|R|-b}}{2^{|R|}} = 2^{-b}.$$

2.5 Conclusion

Thus, with our choice of S from the previous section,

$$\Pr[S \geq 10(m+b)] \leq 2^{-b}.$$

In particular, taking $b = 10$, we obtain

$$\Pr[S \geq 10m + 100] \leq 2^{-10}.$$

Therefore, with high constant probability, the total number of calls to Fix is $O(m)$, and Moser's algorithm terminates in polynomial time.

3 Markov Chains and Random Walks

The study of random walks is important to a lot of areas, particularly complexity theory. There are many interesting results on derandomizing algorithms.

Definition 2 (Time-Homogeneous Markov Chain). *Let Ω be a finite set of states. A sequence of random variables $\{X_t\}_{t\geq 0}$ with $X_t \in \Omega$ is called a* time-homogeneous Markov chain *if it satisfies:*

(Markov Property) *For all $t \geq 0$ and all $x_0, \ldots, x_t, y \in \Omega$,*

$$\Pr(X_{t+1} = y \mid X_0 = x_0, \ldots, X_t = x_t) = \Pr(X_{t+1} = y \mid X_t = x_t).$$

That is, the future is independent of the past given the present.

(Time-Homogeneity) *For all $t \geq 0$ and all $x, y \in \Omega$,*

$$\Pr(X_{t+1} = y \mid X_t = x)$$

does not depend on t.

Equivalently, there exists a matrix $P = (P(x,y))_{x,y\in\Omega}$ such that

$$P(x,y) = \Pr(X_{t+1} = y \mid X_t = x)$$

for all $t \geq 0$. The matrix P is called the transition matrix.

Definition 3 (Transition Matrix). *Given a time-homogeneous Markov chain with state space Ω, the* transition matrix *P is defined by*

$$P_{xy} = \Pr(X_{t+1} = y \mid X_t = x).$$

If π is a probability distribution on Ω written as a row vector, then after one step the distribution of X_{t+1} is given by πP. The matrix P satisfies that $P_{xy} \geq 0$ for all $x, y \in \Omega$, and for every $x \in \Omega$, $\displaystyle\sum_{y\in\Omega} P_{xy} = 1$.

Example 4. *Consider a Markov chain on the state space $\{1,2,3\}$ with transition probabilities given by the following directed graph:*

- *From state 1:*
$$P(1,1) = \tfrac{3}{4}, \qquad P(1,2) = \tfrac{1}{4}, \qquad P(1,3) = 0.$$
- *From state 2:*
$$P(2,1) = 0, \qquad P(2,2) = \tfrac{1}{2}, \qquad P(2,3) = \tfrac{1}{2}.$$

- *From state* 3*:*

$$P(3,1) = \tfrac{1}{2}, \qquad P(3,2) = \tfrac{1}{4}, \qquad P(3,3) = \tfrac{1}{4}.$$

Equivalently, the transition matrix is

$$P = \begin{pmatrix} \frac{3}{4} & \frac{1}{4} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \end{pmatrix}.$$

There are a few different ways to look at Markov Chains. The first one is a stochastic process, where you have a set of random variables taking values in Ω. The interpretation that we care about the most is as a random walk on a directed graph.

Definition 5 (Random Walk on a Graph)**.** *Let $G = (V, E)$ be a directed graph. A* random walk *on G is a time-homogeneous Markov chain $\{S_i\}_{i\geq 0}$ with state space V defined as follows:*

- *$S_0 \in V$ is a chosen start node.*
- *Given the current node $S_i = x$, the next node S_{i+1} is chosen uniformly at random from the set of out-neighbors $N(x) = \{y \in V : (x, y) \in E\}$.*

Let d_x denote the number of edges (out-degree) of x. The transition probabilities are then

$$P(x,y) = \begin{cases} \dfrac{1}{d_x}, & \text{if } (x,y) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

More generally, one can define weighted random walks by assigning probabilities proportional to edge weights, but we will primarily consider the unweighted case.

Given an initial distribution $\pi^{(0)}$, the distribution of the chain after t steps is

$$\pi^{(t)} = \pi^{(0)} P^t.$$

A natural question is whether this sequence of distributions converges as $t \to \infty$, and if so, whether the limit depends on the initial distribution.

Definition 6 (Stationary Distribution)**.** *Let P be the transition matrix of a Markov chain on state space Ω. A probability distribution Π on Ω is called a* stationary distribution *if*

$$\Pi(x) = \sum_{y \in \Omega} \Pi(y) P(y,x) \quad \text{for all } x \in \Omega.$$

Equivalently, in vector notation, $\Pi = \Pi P$. That is, if $X_0 \sim \Pi$, then $X_t \sim \Pi$ for all $t \geq 0$.
A stationary distribution represents a potential long-run equilibrium of the chain.

Below are some properties of certain Markov chains that are useful in characterizing when chains converge to a stationary distribution.

Definition 7 (Irreducible Markov chain)**.** *A Markov chain with transition matrix P on finite state space Ω is called* irreducible *if for every pair of states $x, y \in \Omega$ there exists an integer $t = t(x, y) \geq 0$ such that*

$$P^t(x,y) > 0.$$

Equivalently, from every state x it is possible to reach every other state y with positive probability in some number of steps. I.e., the underlying directed graph is strongly connected.

Definition 8 (Aperiodic Markov chain). *For a state $x \in \Omega$, define its* period *by*

$$d(x) = \gcd\{\, t \geq 1 : P^t(x,x) > 0 \,\}.$$

A state x is called aperiodic *if $d(x) = 1$. The Markov chain is called* aperiodic *if every state is aperiodic.*

Definition 9 (Ergodic Markov chain). *A Markov chain is called* ergodic *if there exists an integer t^* such that for all $t \geq t^*$ and all $x, y \in \Omega$,*

$$P^t(x,y) > 0.$$

Equivalently, for finite state spaces, a Markov chain is ergodic if and only if it is irreducible and aperiodic.

Theorem 10. *If a finite Markov chain is irreducible, then there exists a unique stationary distribution Π^*.*

Example 11. *Let $G = (V, E)$ be a finite, connected, undirected graph, and conside the simple random walk on G. Then the stationary distribution is given by*

$$\Pi^*(v) = \frac{d_v}{2|E|},$$

where d_v is the degree of vertex v and $|E|$ is the number of edges.

In particular:

- *If G is d-regular (i.e., $d_v = d$ for all v), then $\pi^*(v) = \frac{1}{|V|}$, so the stationary distribution is uniform.*
- *More generally, for a directed graph in which every vertex has the same $d = \text{indegree}(v) = \text{outdegree}(v)$ for all v (that is, every vertex has indegree and outdegree d), the uniform distribution is stationary.*
- *This is not true for general directed graphs.*

Definition 12 (Hitting Time). *Let $\{X_t\}_{t\geq 0}$ be a Markov chain on state space Ω. For states $x, y \in \Omega$, the* hitting time *of y starting from x is defined as $\inf\{\, t > 0 : X_t = y \,\}$. The* expected hitting time *from x to y is*

$$h_{xy} = \mathbb{E}[\#\ \textit{steps to go from } x \textit{ to } y].$$

In particular, the recurrence time *(or return time) of a state x is $h_{xx} > 0$.*

Using the notion of hitting times, we can find stationary distributions.

Theorem 13. *Let $\{X_t\}_{t\geq 0}$ be a finite, irreducible Markov chain with stationary distribution Π^*. Fix a state $x \in \Omega$, and let h_{xx} be the first return time to x after time 0. Then*

$$h_{xx} = \frac{1}{\Pi^*(x)}.$$

Proof. Fix $x \in \Omega$. Start the chain in stationarity, i.e. take $X_0 \sim \Pi^*$. Then for every $t \geq 0$ we have $X_t \sim \Pi^*$ as well. By stationarity, $\mathbb{E}[\mathbf{1}\{X_t = x\}] = \Pr(X_t = x) = \Pi^*(x)$ for all t. By the law of large numbers, along a long trajectory, the fraction of time spent in state x converges to $\Pi^*(x)$.

Now think about running the chain for a very long time and just watching when it visits the state x. Every time the chain hits x, you can think of that as the start of a new "cycle." The next time it comes back to x ends that cycle. So the walk is naturally broken into consecutive chunks, where each chunk starts at x and ends the next time we return to x.

Each such chunk has some random length. On average, that length is exactly the expected return time h_{xx}. If we run the chain for a very long time n, then roughly the total time elapsed is about n. The number of visits to x is about

$$\frac{\text{total time}}{\text{average cycle length}} \approx \frac{n}{h_{xx}}.$$

So the fraction of time spent at x is approximately

$$\frac{\#\text{visits to } x}{n} \approx \frac{n/h_{xx}}{n} = \frac{1}{h_{xx}}.$$

But if we start the chain in the stationary distribution, then by definition the long-run fraction of time spent at x must be $\Pi^*(x)$.

Therefore,

$$\Pi^*(x) = \frac{1}{h_{xx}},$$

or equivalently,

$$h_{xx} = \frac{1}{\Pi^*(x)}.$$

□

Note that if the chain is not irreducible, then the stationary distribution may not be unique, e.g., in a graph where the start node leads to two possible chains, which has stationary distributions that are convex combinations of the two sub-chain distributions.

Overall, these stationary distributions are not too difficult to find.

There are some other time-related quantities that we are interested in.

Definition 14 (Total Variation Distance). *Let μ and ν be two probability distributions on a finite state space Ω. The* total variation distance *between μ and ν is*

$$\|\mu - \nu\|_{\text{TV}} = \frac{1}{2} \sum_{x \in \Omega} |\mu(x) - \nu(x)|.$$

Definition 15 (Mixing Time). *Let $\{X_t\}_{t \geq 0}$ be a Markov chain with stationary distribution π^*. For $\varepsilon > 0$, the ε-*mixing time *is defined as*

$$t_{\text{mix}}(\varepsilon) = \min \left\{ t \geq 0 : \max_{x \in \Omega} \|\mathbb{P}_x(X_t \in \cdot) - \pi^*\|_{\text{TV}} \leq \varepsilon \right\}.$$

In words, the mixing time is the smallest time t such that, regardless of the starting state, the distribution of X_t is within ε (in total variation distance) of the stationary distribution.

We'll see mixing times later in the course. Instead, we are going to talk about cover time, which answers a different question.

Definition 16 (Cover Time). *Let $\{X_t\}_{t \geq 0}$ be a Markov chain on a finite state space Ω. Define the* cover time *(starting from an initial distribution π) as*

$$C_\pi = \mathbb{E}_\pi[\min\{\, t \geq 0 : \{X_0, \ldots, X_t\} = \Omega \,\}],$$

i.e., the expected number of steps required to visit every state. For a fixed starting state v, we write

$$C_v = \mathbb{E}_v[\min\{\, t \geq 0 : \{X_0, \ldots, X_t\} = \Omega \,\}].$$

The cover time of the chain is then $C = \max_{v \in \Omega} C_v = \max_\pi C_\pi$.

Note $\mathbb{E}_\pi[X]$ denotes $\mathbb{E}[X \mid X_0 \sim \pi]$, and similarly for probabilities. When we write $\mathbb{E}_v$, this refers to the trivial distribution concentrated at the single starting state v. In general, the cover time can be much larger than the mixing time.

Example 17. *Consider the N-dimensional hypercube graph $\{0,1\}^N$, where two vertices are connected if they differ in exactly one coordinate. This graph has 2^N vertices. For the lazy random walk on the hypercube (that is, a random walk where there is some probability of remaining at the current node at each step), the mixing time is $\Theta(N \log N)$. The cover time, however, must be at least 2^N, since there are 2^N distinct vertices and we must visit each one at least once. Thus the cover time can be exponentially larger than the mixing time.*

Example 18. *Let K_n be the complete graph on n vertices with a self-loop at each vertex. From any current state, the walk chooses each vertex (with replacement) uniformly at random.*

Thus visiting all vertices is exactly the classical coupon collector problem. The expected time to see all n vertices is $C(K_n) = \Theta(n \log n)$.

Example 19. *Let L_n be the path on n vertices (with self-loops). Suppose we start near the middle. After t steps, the typical displacement of the walk is on the order of $\sqrt{t}$ (since it behaves like a simple symmetric random walk). To reach a vertex at distance $\Theta(n)$ from the start therefore requires*

$$\sqrt{t} \approx n \quad \Rightarrow \quad t \approx n^2.$$

Thus the cover time satisfies is $\Theta(n^2)$.

Cover times are already not looking great. But they can get worse!

Example 20. *Consider the* lollipop graph*: a complete graph on $\frac{n}{2}$ vertices connected by a single edge to a path of length $\frac{n}{2}$.*

If the walk starts in the clique, it mixes quickly inside the clique, but the probability of exiting through the single connecting vertex is very small (about $1/n$ per step once at that vertex). It takes on the order of n steps to reach that vertex, and $\Theta(n^2)$ steps before successfully escaping.

Once on the path, covering the path takes $\Theta(n^2)$ time (by the previous example). Putting this together gives a cover time of $\Theta(n^3)$. Yikes!

We can prove that $\Theta(n^3)$ is an upper bound for the cover time on n vertices.

Theorem 21. *For an undirected, connected graph G, $C(G) \leq O(mn)$, where m is the number of edges and n is the number of vertices.*

Proof. Recall that for the simple random walk on an undirected graph, the expected return time to a vertex i is

$$h_{ii} = \frac{2m}{\deg(i)}.$$

We now relate this to hitting times from neighbors of i. Conditioning on the first step from i, we may write

$$h_{ii} = \frac{1}{\deg(i)} \sum_{j \sim i} \bigl(1 + h_{ji}\bigr).$$

From i, we move to a neighbor j in one step, and then need an expected T_{ji} further steps to return to i. Multiplying both sides by $\deg(i)$ gives

$$\deg(i) h_{ii} = \sum_{j \sim i} \bigl(1 + h_{ji}\bigr) = \deg(i) + \sum_{j \sim i} h_{ji}.$$

Using $h_{ii} = \frac{2m}{\deg(i)}$, we obtain

$$2m = \deg(i) + \sum_{j \sim i} T_{ji}.$$

In particular,

$$\sum_{j \sim i} h_{ji} \leq 2m,$$

and therefore for every neighbor j of i,

$$h_{ji} \leq 2m.$$

Now let T be any spanning tree of G. Fix an ordering of the vertices corresponding to a traversal of this tree (for instance, a depth-first traversal). To cover all vertices, it suffices to traverse the spanning tree in this order; the walk is allowed to wander elsewhere, but must eventually realize each edge traversal in the tree.

For each edge (u, v) in the spanning tree, the expected time to go from u to v is h_{uv}. By the bound above, each such hitting time is at most $2m$. Hence for each tree edge,

$$h_{uv} + h_{vu} \leq 4m.$$

Since a spanning tree has $n - 1$ edges, the total expected time to traverse all tree edges in both directions is at most

$$(n - 1) \cdot 4m = O(mn).$$

□

The general $O(mn)$ bound can be sharpened. If G is an undirected, connected graph with minimum degree $\delta = \min_{v \in V} \deg(v)$, then $C(G) = O\big(\frac{mn}{\delta}\big)$ (J. Kahn, N. Linial, N. Nisan, and M. Saks, "On the Cover Time of Random Walks in Graphs).

For directed graphs, the cover time can be exponentially larger. For example, consider a directed line of n vertices

$$1 \to 2 \to 3 \to \cdots \to n,$$

and add an edge from every vertex $i > 1$ back to vertex 1. From each interior vertex, there is a constant probability of being sent back to the start before progressing forward. To reach vertex n, the walk must make $n - 1$ consecutive "forward" moves without being reset to 1. The probability of such a successful run is exponentially small, so the expected time to reach n (and hence to cover the graph) is $\Theta(2^n)$.

4 Preview of Next Steps

Problem 22 (USTCON). *Given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$, determine whether there exists a path from s to t.*

USTCON is especially interesting from the point of view of space minimization. A standard breadth-first search is not space-efficient, since it must store the set of visited vertices, which requires $\Theta(n)$ space.

USTCON can in fact be solved in RL. By this we mean $O(\log n)$ space, polynomial running time, and randomized one-sided error. For randomized algorithms, logarithmic space does not automatically imply polynomial time, so the polynomial-time condition must be stated explicitly. (For deterministic log-space algorithms, one can simply keep a counter and halt after polynomially many steps.)

Since we cannot afford to store the entire explored component, we instead perform a random walk starting at s. At any time, we only need to store the identity of the current vertex, which requires $O(\log n)$ bits, together with a counter for the number of steps taken.

We terminate if we ever reach t, in which case we output `YES`. If we exceed a predetermined number of steps, we output `NO`. From the cover time bound for undirected graphs, we know that $C(G) = O(n^3)$. Thus we run the walk for $2C(G)$ steps.

If t is reachable from s, then with probability at least $1/2$ the walk will have visited every vertex in the connected component of s within $2C(G)$ steps. Indeed, by Markov's inequality,

$$\Pr[\text{time to cover} \geq 2\,\mathbb{E}[\text{time to cover}]] \leq \frac{1}{2}.$$

Thus if s and t are connected, the probability that we fail to see t within $2C(G)$ steps is at most $1/2$. The success probability can be amplified arbitrarily by repeating the procedure independently.

Savitch's theorem shows that many reachability problems can be solved in $O(\log^2 n)$ space. Thus we are particularly interested in situations where one can improve upon this bound.

For directed graphs, the reachability problem (STCON) is complete for NL. Random walks are much less effective in the directed setting, since directed graphs can have exponentially large cover times. Another more meaningful question is to approximate the probability that a random walk starting from s reaches t within a given number of steps. This problem is complete for the complexity class BPL (bounded-error probabilistic log space). A key tool in studying such questions is the theory of expander graphs, which we will develop next.