

Efficient Catalytic Graph Algorithms

James Cook   

Toronto, Canada

Edward Pyne   

MIT, Cambridge, MA, USA

Abstract

We give fast, simple, and implementable catalytic logspace algorithms for two fundamental graph problems.

First, a randomized catalytic algorithm for $s \rightarrow t$ connectivity running in $\tilde{O}(nm)$ time, and a deterministic catalytic algorithm for the same running in $\tilde{O}(n^3m)$ time. The former algorithm is the first algorithmic use of randomization in CL. The algorithm uses one register per vertex and repeatedly “pushes” values along the edges in the graph.

Second, a deterministic catalytic algorithm for simulating random walks which in $\tilde{O}(mT^2/\varepsilon)$ time estimates the probability a T -step random walk ends at a given vertex within ε additive error. The algorithm uses one register for each vertex and increments it at each visit to ensure repeated visits follow different outgoing edges.

Prior catalytic algorithms for both problems did not have explicit runtime bounds beyond being polynomial in n .

2012 ACM Subject Classification Theory of computation \rightarrow Graph algorithms analysis; Theory of computation \rightarrow Pseudorandomness and derandomization; Theory of computation \rightarrow Complexity classes

Keywords and phrases catalytic computing, graph algorithms, catalytic logspace

Digital Object Identifier 10.4230/LIPIcs.ITCS.2026.43

Related Version *Full Version:* <https://arxiv.org/abs/2509.06209>

Funding *Edward Pyne:* Supported by NSF award CCF-2310818.

Acknowledgements E.P. thanks Ryan Williams for encouragement to think about algorithms in CL and useful discussions, and Ian Mertz for the suggestion to work over different moduli. J.C. thanks Michal Koucký and the CCC reviewers for useful suggestions.

1 Introduction

1.1 Catalytic space

In the catalytic space model, an algorithm has two tapes to use as memory: a smaller ordinary tape, and a larger “catalytic” tape which starts out filled with arbitrary data. The algorithm may freely write to and read from both tapes, but when it finishes, the catalytic tape’s original content must be restored. Often the working tape has size $O(\log n)$ and the catalytic tape has size $\text{poly}(n)$, defining the decision class catalytic logspace or CL. Buhrman, Cleve, Koucký, Loff and Speelman [2] initiated the study of this model with their surprising result that evaluation of logarithmic depth threshold circuits was possible in CL, from which it follows for example that $s \rightarrow t$ connectivity and estimation of random walks (the complete problems for nondeterministic logspace (NL) and randomized logspace (BPL) respectively) are in CL.

Broadly speaking, two ways to take advantage of catalytic space have been explored previously: compression-based techniques are able to make use of incompressible catalytic tapes (as randomness, for example); and algebra-based techniques treat the tape as an array



© James Cook and Edward Pyne;

licensed under Creative Commons License CC-BY 4.0

17th Innovations in Theoretical Computer Science Conference (ITCS 2026).

Editor: Shubhangi Saraf; Article No. 43; pp. 43:1–43:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of registers which they use to execute a “straight-line program”, a pre-determined sequence of mathematical operations. The original $\text{TC}^1 \subseteq \text{CL}$ result is an example of the algebraic approach. The compression-based approach began with a “compress-or-random” argument that $\text{BPL} \subseteq \text{CL}$ (Mertz [8] gives a sketch of it). Several subsequent works [6, 10, 5, 7] applied this technique further, with Cook, Li, Mertz and Pyne [5] using it to derandomize randomized CL itself. All of these results in both branches have a “complexity” flavor – in particular, the runtime for the relevant problem is some large polynomial.¹

We give new techniques for deciding graph connectivity and estimating random walks. Our algorithms are very simple and implementable, and allow a precise analysis of the catalytic space consumption and runtime.² We hope this will initiate the study of CL from an algorithmic perspective.

1.2 Our Results: Connectivity

We first state our results for connectivity, beginning with our deterministic algorithm:

► **Theorem 1.** *There is a catalytic algorithm that, given a simple directed graph with n vertices and m edges together with vertices s, t , decides whether there exists a path from s to t . The algorithm uses $O(\log n)$ workspace and $\tilde{O}(n^2)$ catalytic space, and runs in time $\tilde{O}(n^3m)$.*

Next, we give a randomized algorithm that improves the runtime to $\tilde{O}(nm)$ and space usage to $\tilde{O}(n)$, only a factor of n in runtime from the linear-workspace BFS algorithm:

► **Theorem 2.** *There is a randomized catalytic algorithm that, given a simple directed graph with n vertices and $m \geq n$ edges together with vertices s, t , decides whether there exists a path from s to t . The algorithm uses $O(\log n)$ workspace and $\tilde{O}(n)$ catalytic space, and runs in time $\tilde{O}(nm)$.*

We remark that a randomized (one-sided error) catalytic algorithm always resets the catalytic tape no matter the random coins. If there is no $s \rightarrow t$ path, the algorithm always returns no, and if there is an $s \rightarrow t$ path returns yes with probability at least $1/2$.

As far as we know, this result is the first *use* for randomness in catalytic computing – rather than using it to place a new problem in the class (now provably impossible [5]), we use it to give an algorithmic speedup.

► **Remark 3.** The best known algorithms for stconn either use space $O(\log^2 n)$ and super-polynomial time [11], or polynomial time and slightly sublinear space $n/2^{\sqrt{\log n}}$ [1]. Under the assumption there does not exist a randomized stconn algorithm that simultaneously uses $n^{1-\varepsilon}$ space and polynomial time, the catalytic space usage of Theorem 2 is optimal up to subpolynomial factors.³ We view matching the total space of [1] with a catalytic algorithm to be an interesting open question.

In fact, we can obtain an additional desirable property. The practical motivation for CL is to borrow temporarily unused space to perform useful computation. Unfortunately, if any part of the borrowed section of memory is needed during the computation of the

¹ Cook [4] implemented a register program for $s \rightarrow t$ connectivity with an estimated runtime of $\Theta(n^8)$, and catalytic tape size $\Theta(n^3 \log n)$.

² For the runtime bounds, we assume we have RAM access to the catalytic tape and oracle access to the graph. This does not affect the structure of the class (as we can simulate a catalytic RAM with a standard catalytic machine with a polynomial slowdown).

³ A randomized catalytic logspace algorithm for $s \rightarrow t$ connectivity using $n^{1-\varepsilon}$ catalytic space would imply a randomized $n^{1-\varepsilon}$ space, polynomial-time algorithm for $s \rightarrow t$ connectivity [2].

catalytic machine, the original owner may need to wait for the entire catalytic computation to finish. As such, existing catalytic algorithms do not seem to permit sharing a single section of memory without huge latency.⁴ To rectify this problem, we define a notion of catalytic algorithms that must permit fast query access to the *original* memory configuration at all times:

► **Definition 4.** *A catalytic algorithm A is locally revertible in time t if at any point during the execution of A with initial tape τ , the algorithm can be paused and queried on an index i , and will return τ_i in time t (and then continue its execution).*

We show that (at the cost of polynomially more catalytic space) our randomized connectivity algorithm can be made locally revertible for $t = \text{polylog}(n)$:

► **Theorem 5.** *There is a randomized catalytic algorithm that, given a simple directed graph with n vertices and $m \geq n$ edges together with vertices s, t , decides whether there exists a path from s to t . The algorithm uses $O(\log n)$ workspace and $\tilde{O}(n^3)$ catalytic space, and runs in time $\tilde{O}(nm)$. Moreover, the algorithm is locally revertible in time $\text{polylog}(n)$.*

As far as we are aware, no previous catalytic algorithm is locally revertible for any time bound smaller than its runtime. This new property strengthens the motivation for catalytic space: rather than borrowing an unused hard drive, the catalytic algorithm only needs to borrow the ability to *write* to that hard drive, since at all times the original data will be accessible for reading with small latency.

1.3 Our Results: Random Walks

Next, we give an efficient catalytic simulation of random walks:

► **Theorem 6 (random walk on a general graph).** *There is a catalytic algorithm that, given a graph with n vertices and $m \geq n$ edges, together with vertices s, t and parameters $T \in \mathbf{N}, \varepsilon > 0$, returns ρ such that*

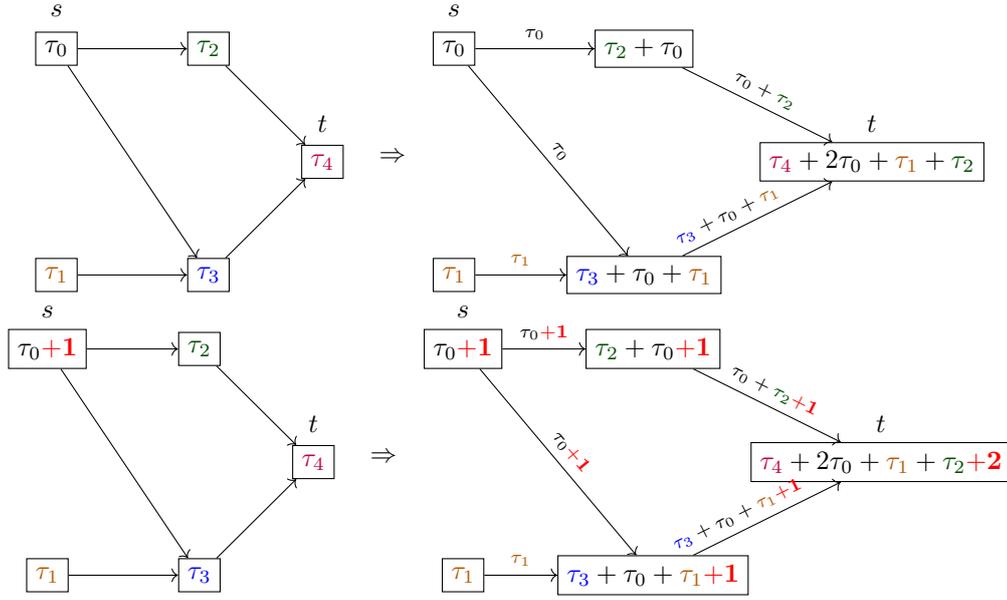
$$|\rho - \Pr[T\text{-step random walk from } s \text{ ends at } t]| \leq \varepsilon.$$

The algorithm runs in time $\tilde{O}(mT^2/\varepsilon)$ and uses $O(\log(mT/\varepsilon))$ workspace and $\tilde{O}(nT \cdot \log(m/\varepsilon))$ catalytic space.

Prior algorithms for random walks are primarily based on logspace derandomization techniques that are not practically efficient. For estimating sub-polynomial length walks to inverse sub-polynomial error, our algorithm runs in almost linear time $m^{1+o(1)}$.

If G is guaranteed to be acyclic, the algorithm can be made to use less time and space by skipping a transformation that removes cycles (see Theorem 21). Alternatively, naïvely applying the algorithm for acyclic graphs on a graph with cycles produces an interesting result: the algorithm is no longer catalytic, but produces a walk with visit counts matching a true random walk's stationary distribution, after a number of steps which depends on the mixing time. For details, see Theorem 29.

⁴ Alongside the large runtime of prior algorithms, this was mentioned as the primary issue for practical catalytic computing by Cook [4].



■ **Figure 1** Pushing values along edges to detect whether a +1 is propagated from s to t .

1.4 Our Technique: Connectivity

We will give a catalytic algorithm that, given a graph $G = (V, E)$ on n vertices, determines if there exists a path from s to t . We virtually lift G to a layered graph on $(n + 1)$ layers with vertex set $\{0, \dots, n\} \times V$, where we place an edge from (i, u) to $(i + 1, v)$ if $(u, v) \in E$.

Next, for every $v \in V$ and timestep $i \in \{0, \dots, n\}$, allocate an $\ell = \tilde{O}(n)$ bit register on the catalytic tape, which we denote $R_{(i,v)}$, and interpret the register as a number in $\mathbf{Z}/2^\ell\mathbf{Z}$. Let the initial value of the register be $\tau_{(i,v)}$.

We define an *edge push* as, for an edge $((i, u), (i + 1, v))$ in the lifted graph, setting

$$R_{(i+1,v)} \leftarrow R_{(i+1,v)} + R_{(i,u)}.$$

For each layer $i = 0, \dots, n - 1$ in sequence, we perform an edge push for every edge in the layer.

Let $\alpha_{(n,t)}$ be the final value of $R_{(n,t)}$ after pushing along every edge. Note that we can easily revert the catalytic tape by performing a *reverse edge push* on every edge in every layer, where we subtract the register instead of adding. Now consider incrementing $R_{(0,s)}$ (i.e. the register of the start vertex in the first layer) by 1 and performing the same sequence of edge pushes; let the new final value of $R_{(n,t)}$ be $\alpha'_{(n,t)}$. We show via an inductive argument that $\alpha'_{(n,t)} - \alpha_{(n,t)}$ is exactly the number of length- n paths from s to t , modulo 2^ℓ (and by adding a self-loop to t we can assume that if an $s \rightarrow t$ path exists, there exists an $s \rightarrow t$ path of length n). By repeatedly executing the edge push sequence and comparing $\alpha_{(n,t)}$ to $\alpha'_{(n,t)}$ bit by bit, we determine whether their difference is nonzero – equivalently, whether there exists an $s \rightarrow t$ path.

Improving the runtime with randomness

Unfortunately, since the number of paths from s to t can be exponential in n and we count the number of paths mod the register size, our deterministic algorithm must take the register size ℓ to be $\Omega(n)$, so pushing a single edge takes linear time. Moreover, we must compute

the entire sequence of pushes $\Omega(\ell/\log n) = \tilde{\Omega}(n)$ times to compare $\alpha_{(n,t)}$ to $\alpha'_{(n,t)}$. To avoid this slowdown, we use randomness. Instead of working mod 2^ℓ for $\ell = \Omega(n)$, we pick a random small modulus q and work mod q . If the number of paths is nonzero, with reasonable probability we will have

$$\alpha_{(n,t)} - \alpha'_{(n,t)} \not\equiv 0 \pmod{q}$$

from which the algorithm can infer that there is a path from s to t . In this way, we reduce the size of each register to $O(\log n)$ bits.

If q is not a power of two, not all strings of length ℓ will correspond to values mod q , so we must ensure that each register is “valid” before running the algorithm. This problem was solved before by Buhrman et. al. [2, Lemma 15], but their approach is not fast enough for us. Instead, we use a larger modulus dq chosen so that $2^\ell - dq$ is small, and then draw a random shift $\beta \in [2^\ell]$ (which we record on the worktape) and add β to every register. With high probability over β , this ensures every register contains a value less than dq (and otherwise we abort).

Local revertibility

Next, we discuss how to achieve local revertibility. After performing all edge pushes onto a register $R_{a=(i+1,v)}$, the current value of this register is exactly its original value τ_a plus

$$\sum_{u:(u,v) \in E} R_{(i-1,u)}$$

where $R_{(i-1,u)}$ is the current value of the register in the previous layer. Thus, if we receive a query for the initial value τ_a of this register, we can iterate over the in-neighbors of v , subtract off the register values of the previous layer, and thus return τ_a . The time for this operation is essentially the product of the in-degree of v and register size. Since we have already reduced the register size to $O(\log n)$, it suffices to ensure our graph has bounded in-degree, which we show we can do with a standard transformation (Lemma 12).

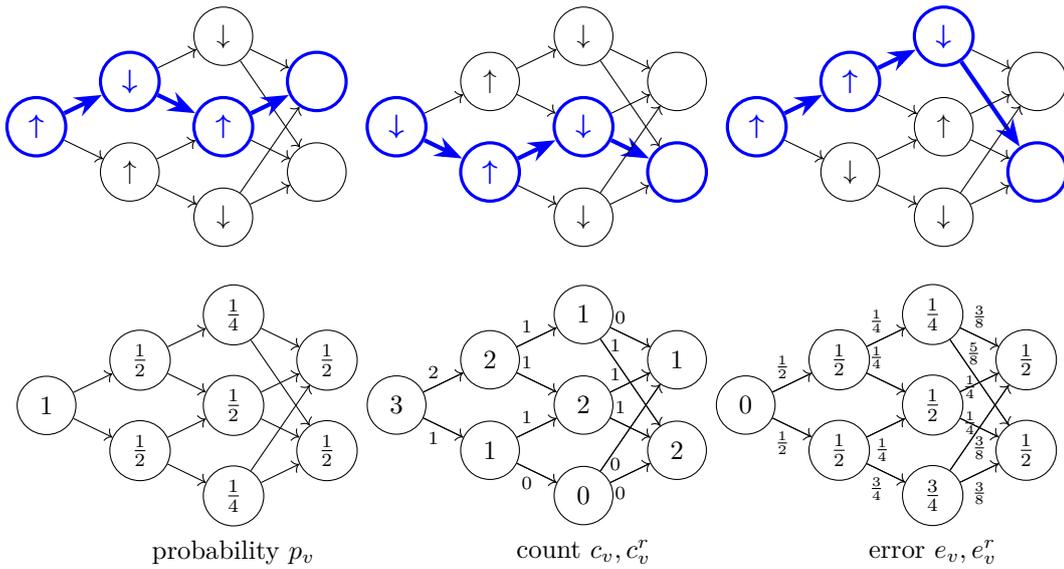
Decreasing the number of registers

Finally, as written we use n^2 registers, one per vertex and timestep. In fact, we can simply use two sets of registers, one for odd and one for even timesteps, and alternate. This saves a factor of n in space, but breaks local revertibility. The version of our algorithm with this modification is Theorem 18.

1.5 Our Technique: Random Walks

In this introduction, for simplicity, we will assume the graph G is acyclic and 2-outregular, with each vertex having a 0 edge and 1 edge. We will determine the probability that a random walk from s ends at t with additive error at most ε .

Allocate one bit of the catalytic tape for each vertex. For vertex v , denote this register R_v . Run $K = \lceil 2m/\varepsilon \rceil$ walks from s as follows. At vertex v , we take the edge labeled with the current value of R_v , then set $R_v \leftarrow 1 - R_v$. In this way, if we examine walks reaching v over the course of the K walks, the next edges taken are $0, 1, 0, 1, \dots$ or $1, 0, 1, 0, \dots$. The top row of Figure 2 shows an example of this process, with the bits of the catalytic tape drawn directly on the corresponding vertices as \uparrow or \downarrow .



■ **Figure 2** Top row: simulated random walks, using one bit of catalytic space per vertex (shown as \uparrow or \downarrow) to alternate between random choices. Bottom row: a comparison to a true random walk, illustrating the parts of Definition 22: visit probabilities p_v (left), visit c_v and transition c_v^r counts (centre), and errors $e_v = |c_v - 3p_v|$, $e_v^r = |\frac{3}{2}p_v|$ (right).

Now we can argue that the number of visits to each vertex approximately equals the expected number of visits if we had done a truly random walk. A common way to prove this kind of result is with a “local consistency check” – see for example Nisan’s Lemma 2.6 [9, 3].

The general idea is that if every vertex is given a pseudorandom bit of 0 approximately as many times as it is given a 1, then each vertex is visited approximately the right number of times. Our version of this argument appears in the proof of Lemma 25. By a careful analysis we show that the number of visits to each vertex is within $2m$ of its expected value, regardless of the number of simulations. Thus after K simulations, we obtain additive error at most $2m/K \leq \epsilon$.

► **Remark 7.** Our runtime is linear in $1/\epsilon$, which is **better** than the $1/\epsilon^2$ dependence of the algorithm that simply takes true random walks. For some intuition on how the algorithm achieves this, note that for every vertex that has been visited T times, every out-edge is visited $T/d \pm 1$ times, whereas for a true random walk, we would expect a deviation on the order of $T/d \pm \sqrt{T}$.

Finally, we must restore the catalytic tape. This is done by running the “reverse” of our K simulations. We do not literally walk in reverse; rather, we walk forward as before, but slightly change the way we supply random bits, so that each “reverse” walk exactly undoes the effect of one normal walk. For details, see Algorithm 2 and Corollary 27.

1.6 Summary of Contributions

From a complexity perspective, neither result places new problems in catalytic logspace. However, we view these algorithms as having two main advantages over prior work. First, the techniques are simple and clearly demonstrate the power of catalytic computation. Second, this simplicity allows us to give concrete bounds on their resource usage (both catalytic space and time). We view *algorithms* in catalytic space as worthy of further study; we have no reason to suspect our runtimes are optimal. In addition, we are interested in whether other catalytic algorithms can be made locally revertible.

2 Preliminaries

We denote $\mathbf{N} = \{0, 1, 2, \dots\}$ the natural numbers, and for $n \in \mathbf{N}$ we denote $[n] = \{0, 1, \dots, n-1\}$ the natural numbers less than n .

For a vertex v in a directed graph, $d_{\text{in}}(v)$ is the number of incoming and $d_{\text{out}}(v)$ is the number of outgoing edges.

We require a very weak bound on the number of paths in a graph:

► **Fact 8.** *For an n -vertex graph (possibly with self-loops), the number of length- T paths between any two vertices is at most $P_n = n^T$.*

We assume basic familiarity with word-RAM and catalytic machines. For concreteness, we give a (not entirely formal) definition of catalytic RAM machines.

► **Definition 9** (Catalytic RAM Machine). *We say \mathcal{A} is a catalytic RAM machine that computes a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ using $T(n)$ time, $S(n)$ workspace, and $W(n)$ catalytic space if it works as follows. The machine is given read-only access to x , read-write access to $S(|x|)$ bits of workspace, read-write access to a catalytic tape R of length $W(|x|)$ in initial configuration τ , and write-only access to an output tape.*

Furthermore, we allow the algorithm query access to the catalytic tape, in that it can read and write a specified bit in constant time after writing the index of this bit to a dedicated query tape. For every x and τ , we have that the machine halts in at most $T(|x|)$ steps with $f(x)$ on the output tape and the catalytic tape restored to τ .

Our definition is not powerful enough to allow analysis of runtimes without polylog factors from query overheads, and we do not attempt this.

2.1 Catalytic Registers

It is often convenient for an algorithm to view its catalytic space as consisting of registers R_1, R_2, \dots for doing arithmetic over some modulus q . If q is a power of two, this is straightforward: allocate $\log q$ bits per register. However, our faster randomized connectivity algorithms (Theorems 2 and 5) need to work with arbitrary moduli q , which creates a difficulty: if we allocate $\ell = \lceil \log q \rceil$ bits of the catalytic tape to each register, some registers may start with values outside the range $0, \dots, 2^\ell - 1$.

Buhrman, Cleve, Koucký, Loff and Speelman [2, Lemma 15] have a clever solution to this problem which unfortunately is too slow for our purposes. Instead, we do the following.

Choose ℓ to be a constant factor larger than $\lceil \log q \rceil$, and treat the ℓ -bit string as an element of $\mathbf{Z}/qd\mathbf{Z}$, where $d = d(q, \ell)$ is the largest value such that $qd \leq 2^\ell$. We say R is *valid* (modulus) q if its initial value is less than qd , and otherwise we say R is *invalid*.

Our algorithm ensures its registers are valid by applying a *shift* to all registers simultaneously.

► **Fact 10.** *For an ℓ -bit register R with initial configuration τ , over a uniformly random shift $\beta \in [2^\ell]$, $(\tau + \beta) \bmod 2^\ell$ is valid for q with probability greater than $1 - 1/(d + 1)$.*

Each valid register can be decomposed as having value $aq + b$; to do arithmetic modulo q , we leave a fixed and only change the b part. It is easy to see these operations can be computed in simultaneous time $\tilde{O}(\ell)$ and space $O(\log \ell)$, given q, d as input. For valid registers R, R' over modulus q , we let $R \leftarrow R \pm R'$ to be this operation.

Now that we can implement registers over arbitrary moduli, the following straightforward lemma lets us use them to check whether counts over a much larger domain are nonzero:

► **Lemma 11.** *Let $V \leq 2^P$ be arbitrary. For a uniformly random $r \in [P^2]$, the probability that $V \equiv 0 \pmod{r}$ is at most $1 - O(1/\log P)$.*

Proof. Note that V can have at most P distinct prime factors, and for every prime q that is not one of these factors we have that $V \not\equiv 0 \pmod{q}$. Moreover, a random element in $[S]$ is prime with probability $\Omega(1/\log S)$ by the Prime Number Theorem. Thus, if we consider the interval $[P^2]$, this interval contains at least $P^2/O(\log P)$ primes, of which at most P divide V , so the probability that we draw a prime that does not divide V is at least

$$\frac{P^2/O(\log P) - P}{P^2} = \Omega(1/\log P). \quad \blacktriangleleft$$

2.2 Input Representation

Because our catalytic algorithms do not have the space to perform otherwise standard transformations on the graph (for instance, producing an adjacency list given an adjacency matrix), we must be careful with how they access the input. We adopt the model of oracle access, which is common in sublinear and local models. We say we have oracle access to a graph G if its vertex set is $[n]$ for some $n \in \mathbb{N}$, and for any $v \in [n]$ we can make the following queries:

- $\text{InDeg}_G(v)$ (resp. $\text{OutDeg}_G(v)$) returns the in-degree $d_{\text{in}}(v)$ (resp. out-degree $d_{\text{out}}(v)$).
- $\text{InNbr}_G(v, i)$ (resp. $\text{OutNbr}_G(v, i)$) returns the i th in-neighbor (resp. out-neighbor) of v , or \perp if this does not exist.

Our connectivity algorithms only use in-edge access to the graph, and our random walk algorithms only use out-edge access.

For the locally revertible connectivity algorithm (Theorem 5), we use that given a graph, we can provide oracle access to a modified graph with bounded in-degree:

► **Lemma 12.** *Given oracle access to a directed graph G with n vertices and m edges, where every vertex has in-degree at most n , there is a simulation of an oracle for a graph G' on $n' = O(n^2)$ vertices with the following properties:*

- *The simulation can answer queries in $O(\log n)$ space and $O(\log n)$ time, with the exception of $\text{OutNbr}_{G'}$ queries.*
- *The maximum in-degree is 2.*
- *The diameter is $\tilde{O}(n)$.*
- *For vertices $s, t \in [n]$, there is an $s \rightarrow t$ path in G if and only if there is an $s \rightarrow t$ path in G' . (We have $[n] \subseteq [n']$, so integers s, t representing vertices of G also represent vertices of G' .)*

All but $O(m)$ of the vertices of G' are isolated (no in- or out-edges), and there is an algorithm to list all non-isolated vertices in $O(\log n)$ space and $\tilde{O}(m + n)$ time.

We defer the proof to the full version of this paper.

3 Deciding Graph Connectivity

We state our first algorithm, which requires one register per vertex and timestep (but permits local revertibility).

► **Theorem 13.** *There is a catalytic logspace algorithm that, given $\ell, q, T \in \mathbf{N}$ and a graph G with n vertices and m edges and vertices s, t , and ℓ -bit registers $R_{i,u}$ for $i \in \{0, \dots, T\}$ and $u \in V$ that are valid for modulus q (Section 2.1), returns*

$$(\# \text{ of } s \rightarrow t \text{ paths of length } T) \bmod q.$$

Moreover, the algorithm runs in time $\tilde{O}(\ell^2 \cdot T \cdot (m + n))$, and is locally revertible in time $\tilde{O}(\ell \cdot d_{\max})$, where d_{\max} is the maximum in-degree of G .

Proof. We first implicitly lift $G = (V, E)$ to a layered graph on $(n + 1)n$ vertices, defined as follows. We let the vertex set be $\{0, \dots, n\} \times V$, and the edge set be

$$E' = \{((i, v), (j, u)) : j = i + 1 \text{ and } (v, u) \in E\}.$$

For every vertex $a = (i, v)$, let R_a be the corresponding register with initial value τ_a .

We then describe the basic algorithm. For an edge $a = (i, u), b = (i + 1, v)$, we let an *edge push* (resp. *reverse edge push*) be the update where we set

$$R_b \leftarrow R_b + R_a \quad (\text{resp. } R_b \leftarrow R_b - R_a)$$

where the arithmetic operations are defined as in Section 2.1.

We let LAYERPUSH_i (resp. REVLAYERPUSH_i) be the operation where we perform an edge push (resp. reverse edge push) on every edge from layer i to layer $i + 1$. We do this by iterating over vertices v , and using $\text{InNbr}_G(v)$ oracle queries to determine the in-neighbors of v , and then pushing along these edges.

Finally, let $\text{INCSTART}(b)$ be the operation where we set

$$R_{(0,s)} \leftarrow R_{(0,s)} + b.$$

For each b , we define the push and reverse sequence

$$\mathcal{P}_b = (\text{INCSTART}(b), \text{LAYERPUSH}_0, \dots, \text{LAYERPUSH}_{T-1})$$

$$\mathcal{R}_b = (\text{REVLAYERPUSH}_{T-1}, \dots, \text{REVLAYERPUSH}_0, \text{INCSTART}(-b)).$$

First, note that the catalytic tape is easy to reset:

▷ **Claim 14.** For every value b and register R_a , after executing $(\mathcal{P}_b, \mathcal{R}_b)$ we have that $R_a = \tau_a$.

Proof. It clearly suffices to prove that $\text{LAYERPUSH}_i, \text{REVLAYERPUSH}_i$ preserve the tape configuration. This follows directly from the linearity of addition and the definition of both operations. ◁

Finally, the difference in the final values at a register with $b = \{0, 1\}$ is exactly the number of paths from s to this register:

► **Lemma 15.** *For every i, v, b , let $\alpha_{(i,v),b}$ be the value of $R_{(i,v)}$ after \mathcal{P}_b . Then $\alpha_{(i,v),1} - \alpha_{(i,v),0}$ is exactly the number of length i $s \rightarrow v$ paths in G , modulo q .*

Proof. Note that $\alpha_{(i,v),b}$ is the value of $R_{(i,v)}$ after

$$\text{INCSTART}(b), \text{LAYERPUSH}_0, \dots, \text{LAYERPUSH}_{i-1}$$

since subsequent operations in \mathcal{P}_b do not write to $R_{(i,v)}$.

43:10 Efficient Catalytic Graph Algorithms

Suppose the claim holds for every vertex in layer $i - 1$. Next, fix an arbitrary vertex $w = (i, v)$, and let $(u_1, \dots, u_r) \subseteq [V]$ be the in-neighbors of v in G . Note that every length- i path from s to v decomposes as

$$(s, \dots, u_j)(u_j, v)$$

for some unique u_j , so the paths are in one-to-one correspondence with length $i - 1$ paths from s to u_j for some j .

Finally, recall that w has in-neighbors

$$a_1 = (i - 1, u_1), \dots, a_r = (i - 1, u_r).$$

and observe that

$$\alpha_{w,b} \equiv \tau_w + \sum_{j \in [r]} \alpha_{a_j,b} \pmod{q}$$

and hence

$$\alpha_{w,1} - \alpha_{w,0} \equiv \sum_{j \in [r]} (\alpha_{a_j,1} - \alpha_{a_j,0}) \pmod{q}$$

and so by the inductive hypothesis we are done. \blacktriangleleft

Then the final algorithm is straightforward. We determine and print $\alpha_{(n,t),1} - \alpha_{(n,t),0} \pmod{q}$. We compute this value by comparing the ℓ -bit registers bit by bit, each time using the sequence $\mathcal{P}_b, \mathcal{R}_b$ with alternating values of b .

► **Lemma 16.** *The algorithm runs in $\tilde{O}(\ell^2 T \cdot (n + m))$ time.*

Proof. Given an edge e and layer i , pushing along this edge takes time $\tilde{O}(\ell)$, and hence a layer push takes time $\tilde{O}((m+n) \cdot \ell)$. Therefore executing \mathcal{P}_b and \mathcal{R}_b takes time $\tilde{O}(T \cdot (n+m)\ell)$. Finally, we invoke both routines ℓ times to compute the final value, so the total runtime is as claimed. \blacktriangleleft

Finally, we show how the algorithm is locally revertible.

► **Lemma 17.** *The algorithm is locally revertible in time $\tilde{O}(\ell \cdot d_{\max})$.*

Proof. Suppose we receive a query to return τ_a , the initial value of the register $R_{a=(i,v)}$. Let u_1, \dots, u_r be the in-neighbors of v , which we can enumerate over using the oracle for G . The register R_a is either in its initial state (in which case we can return its current value without modifying the tape), or some push sequence \mathcal{P}_b has been executed. In that case, the current value of R_a is

$$\begin{aligned} R_a &= \alpha_{a,b} \\ &= \tau_a + \sum_{j \in [r]} \alpha_{(i-1, u_j), b} \\ &= \tau_a + \sum_{j \in [r]} R_{(i-1, u_j)} \end{aligned}$$

where the second equality follows from the definition of \mathcal{P}_b , and the third follows from the fact that we do not modify registers in layer $i - 1$ before reverting layer i to the original

register configuration. Thus, we can recover τ_a by enumerating over the in-neighbors of v and computing

$$R_a - \sum_{j \in [r]} R_{(i-1, u_j)} = \tau_a.$$

Afterwards, we revert R_a to $\alpha_{a,b}$ and continue execution as before. The time for the query is bounded by $\tilde{O}(\ell \cdot r) = \tilde{O}(\ell \cdot d_{\max})$ as claimed. ◀

This completes the proof of the desired properties. ◀

Next, we present a version of the algorithm that avoids lifting the graph, at the cost of not permitting local revertibility. In this case, we do not compute the number of $s \rightarrow t$ paths mod the register size, simply a number that is nonzero if a path exists.

► **Theorem 18.** *For every $T \in \mathbf{N}$ and simple directed graph G with n vertices and m edges and vertices s, t , there is $\zeta_{G,s,t} \in \mathbf{N}$ where:*

- $\zeta_{G,s,t} \leq (n+1)^T$, and
- $\zeta_{G,s,t}$ is nonzero if and only if there is an $s \rightarrow t$ path in G of length at most T .

Moreover, there is a catalytic logspace algorithm that, given $\ell, q, T \in \mathbf{N}$ and a graph G and vertices s, t , and ℓ -bit registers $R_{\sigma,v}$ for $\sigma \in \{0, 1\}$ and $v \in V$ that are valid for modulus q , returns

$$\zeta_{G,s,t} \bmod q.$$

Moreover, the algorithm runs in time $\tilde{O}(\ell^2 \cdot T \cdot (n+m))$.

Proof. We describe the basic algorithm, which is like that of Theorem 13 but uses fewer registers. For each vertex v , the two registers $R_{0,v}, R_{1,v}$ replace the $T+1$ registers $R_{i,v}$ from the previous algorithm.

We define layer push operations given a parity value $\sigma \in \{0, 1\}$. For $(u, v) \in E$ in some fixed order (where we add dummy edges (u, u) for every u), set

$$R_{-\sigma,v} \leftarrow R_{-\sigma,v} + R_{\sigma,u}.$$

and then set $\sigma \leftarrow \neg\sigma$. We define a reverse layer push as, for the same set of edges, setting $\sigma \leftarrow \neg\sigma$ and then

$$R_{-\sigma,v} \leftarrow R_{-\sigma,v} - R_{\sigma,u}.$$

Next let $\text{INCSTART}(b)$ be the operation where we set

$$R_{0,s} \leftarrow R_{0,s} + b.$$

For each b , we initialize $\sigma = 0$ and define the push and reverse sequence:

$$\mathcal{P}_b = (\text{INCSTART}(b), \text{LAYERPUSH}^{(n)}), \quad \mathcal{R}_b = (\text{REVLAYERPUSH}^{(n)}, \text{INCSTART}(-b)).$$

By essentially the same argument as Claim 14, we have that after executing $(\mathcal{P}_b, \mathcal{R}_b)$ for $b \in \{0, 1\}$, every register $R_{\sigma,v}$ is reset to its initial configuration $\tau_{\sigma,v}$. The runtime is straightforward from the description.

Finally, we must prove correctness. For every i , let $\sigma_i \in \{0, 1\}$ be the parity of the registers pushed to in phase i (and note that $\sigma_0 = 0$).

43:12 Efficient Catalytic Graph Algorithms

► **Definition 19.** For every $\sigma \in \{0, 1\}$, vertex v , and integer $i \geq -1$, define $\zeta_{(\sigma,v),i}$ recursively as follows.

- $\zeta_{(1,v),-1} = 0$ for all v .
- $\zeta_{(0,s),0} = 1$ and for $v \neq s$, $\zeta_{(0,v),0} = 0$.
- For every v ,

$$\zeta_{(\sigma_{i+1},v),i+1} := \zeta_{(\sigma_{i-1},v),i-1} + \sum_{u:(u,v) \in E} \zeta_{(\sigma_i,u),i}.$$

We prove that these values are exactly the register difference after the pushes:

► **Lemma 20.** Let $\alpha_{(\sigma_i,v),i,b}$ be the value of $R_{\sigma_i,v}$ after $\text{INCSTART}(b), \text{LAYERPUSH}^{(i)}$. For every $v \in V$ we have

$$\alpha_{(\sigma_i,v),i,1} - \alpha_{(\sigma_i,v),i,0} \equiv \zeta_{(\sigma_i,v),i} \pmod{q}.$$

Proof. For convenience, we define $\alpha_{(1,v),-1,b}$ to be the initial value of $R_{1,v}$ for each vertex v . We prove the lemma by induction.

For the base cases, we have that

$$\alpha_{(1,v),-1,1} - \alpha_{(1,v),-1,0} = 0 = \zeta_{(0,v),-1}$$

$$\alpha_{(0,v),0,1} - \alpha_{(0,v),0,0} = \mathbb{I}[v = s] = \zeta_{(0,v),0}$$

Now assume this holds for i and $i - 1$ and consider the $i + 1$ st push. WLOG suppose $\sigma_{i+1} = 0$. For $v \in V$ we have

$$\alpha_{(0,v),i+1,b} \equiv \alpha_{(0,v),i-1,b} + \sum_{u:(u,v) \in E} \alpha_{(1,u),i,b} \pmod{q}$$

and hence

$$\alpha_{(0,v),i+1,1} - \alpha_{(0,v),i+1,0} \equiv \zeta_{(0,v),i-1} + \sum_{u:(u,v) \in E} \zeta_{(1,u),i} \equiv \zeta_{(1,v),i+1} \pmod{q}. \quad \blacktriangleleft$$

Then a simple inductive argument proves that $\zeta_{(\sigma_i,v),i} \leq (n + 1)^i$, and moreover that the set of v for which $\zeta_{(\sigma_i,v),i} > 0$ is exactly those v with an $s \rightarrow v$ path of length at most i . \blacktriangleleft

Putting it all together

We then use these results to prove the main theorems.

For the deterministic algorithm, we choose the register size and modulus q large enough so that the count of paths mod q is equal to the count of paths.

► **Theorem 1.** There is a catalytic algorithm that, given a simple directed graph with n vertices and m edges together with vertices s, t , decides whether there exists a path from s to t . The algorithm uses $O(\log n)$ workspace and $\tilde{O}(n^2)$ catalytic space, and runs in time $\tilde{O}(n^3m)$.

Proof. We initialize $2n$ registers $\{R_{\sigma,v}\}_{\sigma \in \{0,1\}, v \in V}$ each of size $\ell = \lceil \log((n + 1)^n) + 1 \rceil$ and set $q = 2^\ell$. Since we chose $q = 2^\ell$, all registers are valid no matter their initial configuration, so we immediately invoke Theorem 18 with $T = n$. If the value obtained is nonzero, we return that there is a path, and otherwise return that there is no path. The runtime is immediate from the choice of ℓ and T . \blacktriangleleft

We now give the randomized algorithm.

► **Theorem 2.** *There is a randomized catalytic algorithm that, given a simple directed graph with n vertices and $m \geq n$ edges together with vertices s, t , decides whether there exists a path from s to t . The algorithm uses $O(\log n)$ workspace and $\tilde{O}(n)$ catalytic space, and runs in time $\tilde{O}(nm)$.*

Proof. Let $B_n = (n + 1)^n + 1$.

The Algorithm. For $I = O(\log n)$ iterations (where the specific constant is to be chosen later), we proceed as follows. We initialize $2n$ registers $\{R_{\sigma,v}\}_{\sigma \in \{0,1\}, v \in V}$, each of size

$$\ell = 5 \lceil \log B_n \rceil.$$

We draw a random modulus $q \in [\log^2 B_n]$ and store it on the worktape. Let d be the largest value such that $qd \leq 2^\ell$ (which we can compute in time $\text{polylog}(n)$ and store on the worktape). We have that

$$d \geq \frac{2^\ell}{2q} > n^3$$

Next, we draw a random shift $\beta \in [2^\ell]$ and store it on the worktape. We add β to each register $R_{\sigma,v}$ with initial configuration $\tau_{\sigma,v}$ and verify that $\tau_{\sigma,v} + \beta$ is valid for q . If not, we subtract β from all registers and abort (and return \perp). Otherwise, we invoke Theorem 18 with this register set and $T = n$. Let the returned value be ζ . We first subtract β from all registers. Then, if $\zeta \neq 0$, we return that there is a path, and otherwise proceed to the next iteration. If we exhaust all iterations, we return that there is not a path.

Success Probability. We first argue that the algorithm does not abort with high probability. There are $2n$ registers, each of which we shift $O(\log n)$ times. By Fact 10, the probability that any such shift results in an invalid configuration is at most $2/n^3$, so a union bound completes the proof.

Next, note if $\zeta_{G,s,t} = 0$, the algorithm clearly never returns there is a path. Otherwise, for each q drawn by the algorithm, by Lemma 11 the probability that $\zeta_{G,s,t} \equiv 0 \pmod{p}$ is at most $(1 - O(1/\log \log B_n)) = (1 - 1/O(\log n))$, and hence choosing $I = O(\log n)$ sufficiently large, we obtain that with probability at least $1 - 1/100$ there is some iteration where we detect a nonzero number of paths, and thus succeed.

Finally, runtime and space consumption follow directly from the description of the algorithm. ◀

Finally, we finish the proof of the locally revertible algorithm.

► **Theorem 5.** *There is a randomized catalytic algorithm that, given a simple directed graph with n vertices and $m \geq n$ edges together with vertices s, t , decides whether there exists a path from s to t . The algorithm uses $O(\log n)$ workspace and $\tilde{O}(n^3)$ catalytic space, and runs in time $\tilde{O}(nm)$. Moreover, the algorithm is locally revertible in time $\text{polylog}(n)$.*

Proof. We first modify the input graph G by simulating access to the graph G' using the query oracle of Lemma 12. Let $n' = O(n^2)$ be the number of vertices of G' , let $n'' = O(m)$ be the number of non-isolated vertices, let $T = \tilde{O}(n)$ be its diameter, and recall that (after virtually adding a self-loop on t) it has maximum in-degree 3. Let $p_{n'} = \lceil \log P_{n'} \rceil$ be the logarithm of the bound on the number of paths in G' of Fact 8.

43:14 Efficient Catalytic Graph Algorithms

The Algorithm. For $I = O(\log n)$ iterations (where the specific constant is to be chosen later), we proceed as follows. First, we initialize the registers we will use by adding a random shift as described in Section 2.1. We will have a register corresponding to each layer and vertex in G' , indexed as

$$(i, v) \in \{0, \dots, T\} \times [n'].$$

We allocate n^3 total registers on the catalytic tape corresponding to these vertices, but we only initialize the $Tn'' = \tilde{O}(nm)$ registers which correspond to non-isolated vertices of G' . (Recall Lemma 12 gives an algorithm for enumerating these vertices in $\tilde{O}(m+n)$ time, amounting to $\tilde{O}(n(m+n))$ time for n layers.) Call a register *relevant* if it corresponds to (i, v) where v is a non-isolated vertex or $v \in \{s, t\}$.

Each register has size

$$\ell = 5 \lceil \log p_{n'} \rceil.$$

We draw a random modulus $q \in [p_{n'}^2]$ and store it on the worktape. Let d be the largest value such that $qd \leq 2^\ell$ (which we can compute in time $\text{polylog}(n)$ and store on the worktape). We have that

$$d \geq \frac{2^\ell}{2q} \geq \frac{m^3}{2}.$$

Next, we draw a random shift $\beta \in [2^\ell]$ and store it on the worktape. We add β to each relevant register R_a with initial configuration τ_a , and then verify that each $\tau_a + \beta$ is valid for q . If not, we first subtract β from all relevant registers, and then abort (and return \perp). Otherwise, we invoke Theorem 13 with

$$G = G', \quad T = T$$

except we modify it to only push values from relevant registers. Let the returned value be α . We restore the catalytic tape by subtracting β from all relevant registers. Then, if $\alpha \neq 0$, we return there is a path, and otherwise proceed to the next iteration. If we exhaust all iterations, we return that there is not a path.

Success Probability. We first argue that the algorithm does not abort with high probability. Note that there are $\tilde{O}(nm)$ registers, each of which we shift $O(\log n)$ times. By Fact 10, the probability that any such shift results in an invalid configuration is at most $4/m^3$, so a union bound completes the proof.

Next, let V be the total number of length- n paths from s to t in the modified graph. If $V = 0$, the algorithm clearly never returns there is a path. Otherwise, for each q drawn by the algorithm, by Lemma 11 the probability that $V \equiv 0 \pmod{p}$ is at most $(1 - O(1/\log p_{n'})) = (1 - 1/O(\log n))$, and hence choosing $I = O(\log n)$ sufficiently large, we obtain that with probability at least $1 - 1/100$ there is some iteration where we detect a nonzero number of paths, and thus succeed.

Finally, runtime and space consumption follow directly from the description of the algorithm.

Local Revertibility. Finally, we argue that the algorithm is locally revertible. Given a query on register a , we query the local revertibility routine of Theorem 13 on this register. Since we initialized the algorithm of Theorem 13 with R_a in configuration $\tau_a + \beta$, it returns this value, and we return τ_a by subtracting the stored shift β . Finally, since G' has constant degree, the time is as claimed. \blacktriangleleft

4 Estimating Random Walks

The bulk of our proof of Theorem 6 is a technique for simulating random walks on *acyclic* graphs:

► **Theorem 21** (random walk on a DAG). *There is an algorithm which, given a directed acyclic graph G with n vertices and m edges, together with vertex s , sink vertex t , and parameter $\varepsilon > 0$, returns ρ such that*

$$|\rho - \Pr[\text{random walk from } s \text{ reaches } t]| \leq \varepsilon.$$

The algorithm runs in time $\tilde{O}(nm/\varepsilon)$. It uses $O(\log(nm/\varepsilon))$ workspace and $\tilde{O}(n \log(m/\varepsilon))$ catalytic space; the algorithm is guaranteed to restore the catalytic tape to its initial state as long as the input is valid – in particular, the graph G has no cycles.

(This version of our algorithm has the curious property that it can be tricked into making irreversible changes to its catalytic tape if G has cycles. If this is undesirable, the algorithm could be changed to require an efficiently checkable proof that G is acyclic, like a topological ordering of its vertices.)

The algorithm is described by WALK (Algorithm 1) and its subroutine WALK_ONCE (Algorithm 2). To prove Theorem 21, we must prove two things for every G, s, t, ε : in Section 4.2, we show that WALK(G, s, t, ε) gives the correct output, and in Section 4.3, we show that it restores the catalytic tape at the end of the computation. Section 4.4 ties the proof together and analyzes the time and space used. Section 4.5 proves Theorem 6 by converting any graph into a larger acyclic graph, and then in Section 4.6, we explore what happens if we apply our technique directly to a graph with cycles, skipping the conversion step.

4.1 Registers

The algorithm uses one register R_v on the catalytic tape for every vertex v of the graph. We allocate ℓ bits to each register, where $\ell = \lceil \log K \rceil$ and $K = \lceil 2m/\varepsilon \rceil$ is the number of simulations run by Algorithm 1. So, each register stores a number in the range $0, \dots, 2^\ell - 1$. We choose this value of ℓ so that if we increment the value of a register K times, each time adding one modulo 2^ℓ , there is at most one time when it resets to 0 instead of increasing by one. Concretely, this is used in the proof of Lemma 23 to bound the error introduced by this reset.

4.2 The output is correct

To evaluate how well WALK simulates a random walk on G , we will compare the number of times WALK_ONCE visits each vertex to the probability a true random walk would reach it. We will argue that for every vertex v , the algorithm splits its time fairly over the $d_{\text{out}}(v)$ different outgoing edges when it takes a step on line 11, from which it will follow that our simulation is sufficiently accurate.

Throughout this section, fix G, s, t and ε . Fix an initial content of the catalytic tape τ , and let R_v be the ℓ -bit register allocated to vertex v with initial value τ_v .

Let $K = \lceil 2m/\varepsilon \rceil$ as in WALK.

The following definition establishes some notation to help reason about the accuracy of the simulation. It is illustrated in Figure 2.

■ **Algorithm 1** $\text{WALK}(G, s, t, \varepsilon)$. Parameters: graph G , source s , target t , accuracy ε .

```

1 Let  $K = \lceil 2m/\varepsilon \rceil$ .
2  $n_{\text{reach}} \leftarrow 0$ 
  /* Forward phase: estimate probability of reaching  $t$  */
3 for  $i \in [K]$  do
4    $v \leftarrow \text{WALK\_ONCE}(M, x, s, \text{forward})$ 
5   if  $v = t$  then
6      $n_{\text{reach}} \leftarrow n_{\text{reach}} + 1$ 
7   end
8 end
  /* Reverse phase: reset the catalytic tape */
9 for  $i \in [K]$  do
10   $\text{WALK\_ONCE}(M, x, s, \text{reverse})$ 
11 end
12 return  $n_{\text{reach}}/K$ 

```

■ **Algorithm 2** $\text{WALK_ONCE}(G, s, m)$. Parameters: graph G , source s , mode $m \in \{\text{forward}, \text{reverse}\}$. Returns a sink vertex of G . (This algorithm modifies the catalytic tape, so it is not a catalytic algorithm. However, the changes are reversible, so it can be used as a subroutine in a catalytic algorithm.)

```

1 Registers: one register  $R_v$  in  $[2^\ell]$  for every vertex  $v$ ; see Section 4.1.
2  $v \leftarrow s$ 
3 while  $\text{OutDeg}(v) > 0$  do
  /* Choose an outgoing edge based on  $R_v$ , and update  $R_v$ . */
4   if  $m = \text{forward}$  then
5      $r \leftarrow R_v \bmod \text{OutDeg}(v)$ 
6      $R_v \leftarrow (R_v + 1) \bmod 2^\ell$ 
7   else /*  $m = \text{reverse}$  */
8      $R_v \leftarrow (R_v - 1) \bmod 2^\ell$ 
9      $r \leftarrow R_v \bmod \text{OutDeg}(v)$ 
10  end
11   $v \leftarrow \text{OutNbr}_G(v, r)$ 
12 end
13 return  $v$ 

```

► **Definition 22** (visit probability p_v , visit count c_v , error e_v , transition count c_v^r , transition error e_v^r). Let p_v be the probability that a random walk starting at s reaches a vertex v .

Let c_v be the number of times WALK_ONCE visits v during the forward phase of WALK (lines 3–8). (A “visit” to the vertex stored in the variable v occurs whenever WALK_ONCE evaluates the while loop condition on line 3.)

Define the error $e_v = |c_v - Kp_v|$.

For $r \in [d_{\text{out}}(v)]$, let c_v^r be the number of those visits for which the variable r had the given value on line 11 of WALK_ONCE (so $c_v = \sum_{r=0}^{d_{\text{out}}(v)-1} c_v^r$ unless v is a sink). c_v^r counts transitions where the algorithm followed the r -th outgoing edge from v .

Define the transition error $e_v^r = |c_v^r - Kp_v/d_{\text{out}}(v)|$.

► **Lemma 23.** For every vertex v and $r \in [d_{\text{out}}(v)]$, $e_v^r \leq 2 + e_v/d_{\text{out}}(v)$.

Proof. Let us temporarily imagine the register R_v always stores a value in $[d_{\text{out}}(v)]$, and that each time WALK_ONCE visits v , line 6 increments it as $R_v \leftarrow (R_v + 1) \bmod d_{\text{out}}(v)$ instead of incrementing modulo 2^ℓ . This causes the value r to cycle through the values $0, \dots, d_{\text{out}}(v) - 1$, so that on the t -th visit to v , $r = (\tau_v + t - 1) \bmod d_{\text{out}}(v)$, where τ_v is the starting value of R_v from the catalytic tape. As a result, any two transition counts must differ by at most one:

$$|c_v^r - c_v^{r'}| \leq 1 \quad (\text{pretending } R_v \in [d_{\text{out}}(v)]). \quad (1)$$

In fact, R_v cycles through the values $0, \dots, 2^\ell$. As a result r cycles through the values $0, \dots, d_{\text{out}}(v) - 1$ in order, with one exception: if register R_v cycles from $2^\ell - 1$ to 0, then the cycle is interrupted. Since $2^\ell \geq K$, this can happen at most once. So, instead of Equation (1), we have have that

$$|c_v^r - c_v^{r'}| \leq 2.$$

Since $c_v = \sum_{r=0}^{d_{\text{out}}(v)-1} c_v^r$, it follows that

$$\left| c_v^r - \frac{c_v}{d_{\text{out}}(v)} \right| = \left| c_v^r - \sum_{r'=0}^{d_{\text{out}}(v)-1} \frac{c_v^{r'}}{d_{\text{out}}(v)} \right| \leq \frac{1}{d_{\text{out}}(v)} \sum_{r'=0}^{d_{\text{out}}(v)-1} |c_v^r - c_v^{r'}| < 2.$$

and so

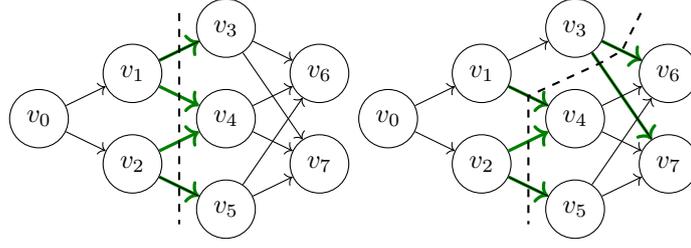
$$e_v^r = \left| c_v^r - \frac{Kp_v}{d_{\text{out}}(v)} \right| \leq \left| c_v^r - \frac{c_v}{d_{\text{out}}(v)} \right| + \left| \frac{c_v}{d_{\text{out}}(v)} - \frac{Kp_v}{d_{\text{out}}(v)} \right| < 2 + \frac{e_v}{d_{\text{out}}(v)}. \quad \blacktriangleleft$$

► **Lemma 24.** Let $(u_1, v), \dots, (u_d, v)$ be all edges incoming to a vertex v , where each (u_i, v) is the r_i -th outgoing edge from u_i . Then $e_v \leq \sum_{i=1}^d e_{u_i}^{r_i}$.

Proof. Using the facts that $c_v = \sum_{i=1}^d c_{u_i}^{r_i}$ and $p_v = \sum_{i=1}^d p_{u_i}/d_{\text{out}}(u_i)$, we have:

$$e_v = |c_v - Kp_v| = \left| \sum_{i=1}^d (c_{u_i}^{r_i} - Kp_{u_i}/d_{\text{out}}(u_i)) \right| \leq \sum_{i=1}^d |c_{u_i}^{r_i} - Kp_{u_i}/d_{\text{out}}(u_i)| = \sum_{i=1}^d e_{u_i}^{r_i} \quad \blacktriangleleft$$

► **Lemma 25.** The final value n_{reach}/K returned by WALK on line 12 is within additive error ε of the true visit probability p_t .



■ **Figure 3** Two steps in the induction argument in the proof of Lemma 25. On the left, $i = 2$ and on the right, $i = 3$. In each case, F_i consists of the edges that cross the dashed line, and σ_i is the sum of the errors on those edges.

Proof. We prove this from Lemmas 23 and 24 using an induction argument.

Let V be the set of vertices of G reachable from the start vertex s , and let $v_0 = s, v_1, \dots, v_n = t$ be a topological order on V . That is, for any edge (u, v) , u appears before v in the order.

For $i \in \{0, 1, \dots, n-1\}$, consider the cut of G with vertices v_0, \dots, v_i on the left and v_{i+1}, \dots, v_n on the right. We are interested in the total error over all the transitions which cross each such cut.

Let $F_i \subseteq \{(v, r) \mid v \in V, r \in [d_{\text{out}}(v)]\}$ be the set of transitions which cross the cut. That is, a pair (v, r) is in F_i if v is on the left of the cut and the r -th outgoing edge from v leads to a vertex on the right of the cut. See Figure 3.

Let $D_i = \sum_{j=0}^i d_{\text{out}}(v_j)$: that is, D_i is the number of edges that originate from the left side of the cut (whether or not they cross the cut).

Let $\sigma_i = \sum_{(v,r) \in F_i} e_v^r$, recalling from Definition 22 that $e_v^r = |c_v^r - Kp_v / d_{\text{out}}(v)|$. We will show by induction that $\sigma_i \leq 2D_i$ for all i .

Base case: $\sigma_0 = \sum_{r=0}^{d_{\text{out}}(v_0)} e_{v_0}^r$. We know $c_{v_0} = K$ and $p_{v_0} = 1$, so $e_{v_0} = 0$ and so by Lemma 23, $\sigma_0 \leq 2d_{\text{out}}(v_0)$.

Induction step: Fix $i \in [n-1]$ and assume $\sigma_i \leq 2D_i$. Let $(v_{j_1}, v_{i+1}), \dots, (v_{j_d}, v_{i+1})$ be the edges incoming to v_{i+1} , where each (v_{j_k}, v_{i+1}) is the r_k -th outgoing edge from v_{j_k} . Those are the transitions that contribute to σ_i but not σ_{i+1} , so we have $\sigma_{i+1} = \sigma_i + \sum_{r=0}^{d_{\text{out}}(v_{i+1})-1} e_{v_{i+1}}^r - \sum_{k=1}^d e_{v_{j_k}}^{r_k}$. From Lemma 23, we have

$$\sum_{r=0}^{d_{\text{out}}(v_{i+1})-1} e_{v_{i+1}}^r \leq 2d_{\text{out}}(v_{i+1}) + e_{v_{i+1}},$$

and so applying Lemma 24 gives

$$\sigma_{i+1} \leq \sigma_i + 2d_{\text{out}}(v_{i+1}) + e_{v_{i+1}} - \sum_{k=1}^d e_{v_{j_k}}^{r_k} \leq \sigma_i + 2d_{\text{out}}(v_{i+1}) \leq 2D_{i+1}$$

completing the induction.

In particular, $\sigma_{n-1} \leq 2m$. The corresponding set of edges F_{n-1} are exactly vertex t 's in-edges. Therefore, by Lemma 24, e_t is at most

$$\sum_{(v,r) \in F_{n-1}} e_v^r = \sigma_{n-1} \leq 2m$$

When WALK reaches line 12, $n_{\text{reach}} = c_t$, and so the algorithm returns c_t/K . This is within ε of p_t because

$$\left| \frac{n_{\text{reach}}}{K} - p_t \right| = \frac{e_t}{K} \leq \frac{2m}{\lceil 2m/\varepsilon \rceil} \leq \varepsilon. \quad \blacktriangleleft$$

4.3 The catalytic tape is restored

WALK restores its catalytic tape when it finishes. To see this, the following is enough:

► **Lemma 26.** *Running WALK_ONCE($G, s, t, \text{forward}$) then WALK_ONCE($G, s, t, \text{reverse}$) leaves all register values R_v with the same values they started with.*

Proof. It is enough to show both calls to WALK_ONCE visit the same sequence of vertices, since then each register is incremented and decremented the same number of times. (We say WALK_ONCE “visits” vertex v each time the loop condition on line 3 is evaluated.)

Loosely speaking, this is true because each time WALK_ONCE($G, s, \text{reverse}$) decides which outgoing edge $\text{OutNbr}_G(v, r)$ to follow, it first (line 8) undoes the change made by WALK_ONCE($G, s, \text{forward}$), and so it ends up choosing the same edge index r . This argument relies on the fact that a single run of WALK_ONCE never modifies the same register R_v more than once, which follows from the fact that G has no cycles.

To make this more precise, let v_0, \dots, v_t be the vertices visited by WALK_ONCE($G, s, \text{forward}$), and $v'_0, \dots, v'_{t'}$ be the vertices visited by the subsequent call to WALK_ONCE($G, s, \text{reverse}$). We show by induction that $v_i = v'_i$ for each i , so that in particular the main loop ends at the same sink vertex in each case and so $t = t'$.

To begin with, $v_0 = v'_0 = s$. Now assume $v_i = v'_i$. If v_i is a sink, both subroutine calls halt and we are finished. Otherwise, we must show the same value r is chosen both times. Since neither subroutine call made any other changes to R_v , when the second subroutine call subtracts one from R_v on line 8, it exactly cancels out the only change made by the first subroutine call on line 6, and so the same value r is recovered, and so the same next step is taken: $v_{i+1} = v'_{i+1}$. ◀

► **Corollary 27.** *WALK leaves its catalytic tape unchanged.*

Proof. This is the same as saying the final values of all registers R_v equal the initial values.

By induction on K , we can see that K calls to WALK_ONCE($G, s, \text{forward}$) followed by K calls to WALK_ONCE($G, s, \text{reverse}$) has no net effect on the registers R_v . For $K = 0$ this is clear. Lemma 26 provides the induction step. That is, $K + 1$ calls to each can be decomposed as (1) K calls to WALK_ONCE($G, s, \text{forward}$), then (2) a single call to each, which by Lemma 26 has no net effect, then (3) K calls to WALK_ONCE($G, s, \text{reverse}$). Since (2) has no effect, we are left with K calls each, which by the induction hypothesis have no net effect. ◀

4.4 Proof of Theorem 21 (random walk on a DAG)

Lemma 25 proves that WALK gives a correct answer, and Corollary 27 proves that it restores its catalytic tape. The runtime is dominated by $2K$ calls to WALK_ONCE, each of which visits each vertex of G at most once. Visiting a vertex means executing one iteration of the main loop of WALK_ONCE, which takes $\text{polylog}(n + m)$ time, so the total run time is $\tilde{O}(nK \text{ polylog } m) = \tilde{O}(nm/\varepsilon)$.⁵ Each register takes $O(\ell) = O(\log(m/\varepsilon))$ bits of the catalytic

⁵ It also uses $\tilde{O}(n \text{ polylog } m)$ time to count the number of edges m in order to compute K .

tape (Section 4.1), for a total of $\tilde{O}(n \log(m/\varepsilon))$ catalytic space. The working memory only includes a constant number of variables n_{reach} , v , etc, each taking $O(\log(nm/\varepsilon))$ working memory. ◀

4.5 Proof of Theorem 6 (random walk on a general graph)

For convenience, we restate the theorem here:

► **Theorem 6** (random walk on a general graph). *There is a catalytic algorithm that, given a graph with n vertices and $m \geq n$ edges, together with vertices s, t and parameters $T \in \mathbf{N}, \varepsilon > 0$, returns ρ such that*

$$|\rho - \Pr[T\text{-step random walk from } s \text{ ends at } t]| \leq \varepsilon.$$

The algorithm runs in time $\tilde{O}(mT^2/\varepsilon)$ and uses $O(\log(mT/\varepsilon))$ workspace and $\tilde{O}(nT \cdot \log(m/\varepsilon))$ catalytic space.

Proof. We construct an acyclic graph $G' = (V', E')$ by creating $T + 1$ copies of the input graph $G = (V, E)$ and arranging them in layers, with edges going from each layer i to layer $i + 1$. That is, $V' = [T + 1] \times V$, and $E' = \{((i, u), (i + 1, v)) \mid t \in [T], (u, v) \in E\}$.

We then run WALK (Algorithm 1) on the graph G' , using start vertex $(0, s)$ and sink vertex (T, t) , and keeping the same parameter ε . Since a random walk on G' from s to a sink is equivalent to a T -step random walk on G , the proof of Theorem 21 implies the algorithm will estimate the probability that a T -step random walk ends at vertex t within the required additive error bound of ε . The same proof also implies our algorithm is catalytic.

The space bounds ($O(\log nmT/\varepsilon)$ working space and $\tilde{O}(nT \log(m/\varepsilon))$ catalytic space) follow directly from the proof of Theorem 21, since G' has nT vertices. The runtime is the time needed to run WALK_ONCE $K = O(mT/\varepsilon)$ times. Each call to WALK_ONCE takes $\tilde{O}(T)$ time, since every walk will have T steps. So, the total runtime is $\tilde{O}(mT^2/\varepsilon)$. ◀

4.6 What if we don't eliminate cycles?

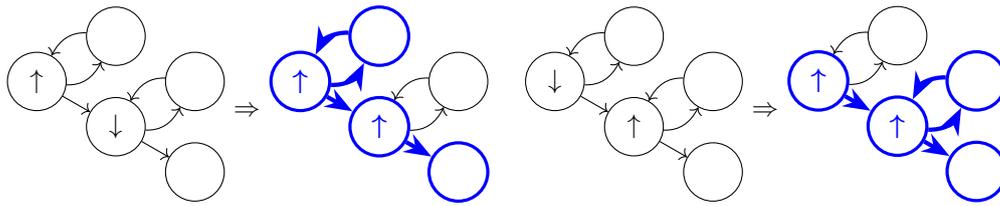
The proof of our algorithm's correctness relies on the graph being acyclic, and so when we are given a general graph, we are forced to pay a penalty in space and time to convert it to an acyclic one. It is tempting to try avoiding the penalty by running the algorithm directly on a graph with cycles.

As it turns out, we end up with an algorithm that is not catalytic, but still accurately simulates a random walk, in the sense that it approaches the graph's stationary distribution: see Theorem 29.

It would be interesting to try to modify this algorithm to be catalytic without incurring a time or space penalty. This seems challenging, because it is possible to construct a graph that causes WALK_ONCE to lose information that was stored on the catalytic tape: two different initializations of the registers R_v lead to the same final register values, and so recovery is impossible. See Figure 4.

To describe the behaviour of this new algorithm, we first define four terms:

► **Definition 28** (probability vector, stochastic matrix, stationary distribution, mixing time). *A probability vector $v \in \mathbf{R}^n$ is any vector with nonnegative entries and $|v|_1 = 1$. A matrix $W \in \mathbf{R}^{n \times n}$ is (left) stochastic if every column is a probability vector. The stationary distribution of a stochastic matrix $W \in \mathbf{R}^{n \times n}$ is any probability vector π such that $W\pi = \pi$. We say W mixes in time T with error ε if the stationary distribution π is unique and for every probability vector, $|W^T v - \pi| \leq \varepsilon$. (Here, W^T is the T -th power of W , not its transpose.)*



■ **Figure 4** An example showing why the algorithm of Theorem 29 can't restore its register values, and so is not catalytic. Two runs of the algorithm for $T' = 4$ steps are shown. The first and third graphs show two different ways to initialize two of the catalytic registers, with \uparrow, \downarrow representing the possible values as in Figure 2. The second and fourth graphs show the resulting walks highlighted in blue, and the final values of those registers. The final register values are the same, so the algorithm cannot know which initial values to restore the registers to.

We remark that our definition of mixing time implies the Markov chain is ergodic.

► **Theorem 29.** *There is an algorithm which, given a graph G with n vertices and m edges together with vertex v^* and parameters $T, \delta \in \mathbf{N}$, returns a number ρ which approximates the stationary probability at v^* in the following sense. If the random walk on G has stationary distribution π and mixes in time T with accuracy ε , then*

$$|\rho - \pi(v^*)| \leq \varepsilon + \delta.$$

The algorithm runs in time $\tilde{O}(Tm/\delta)$ and uses space $\tilde{O}(n)$.

We defer the proof to the full version of this paper.

5 Future Directions

We hope these examples will inspire others to find new efficient catalytic algorithms for these or other problems. In particular, it would be interesting to avoid the overhead in Theorem 6 from converting to an acyclic graph – Theorem 29 attempts this, but the algorithm fails to be catalytic. For connectivity, our algorithms are all incomparable (in speed, randomness, and revertibility). It would be interesting to obtain a best of both worlds result.

References

- 1 Greg Barnes, Jonathan F. Buss, Walter L. Ruzzo, and Baruch Schieber. A sublinear space, polynomial time algorithm for directed s - t connectivity. *SIAM J. Comput.*, 27(5):1273–1282, 1998. doi:10.1137/S0097539793283151.
- 2 Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, STOC '14*, pages 857–866, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2591796.2591874.
- 3 Kuan Cheng and William M. Hoza. Hitting sets give two-sided derandomization of small space. *Adv. Math. Commun.*, 18:1–32, 2022. doi:10.4086/T0C.2022.V018A021.
- 4 James Cook. How to borrow memory. <https://www.falsifian.org/blog/2021/06/04/catalytic/>.
- 5 James Cook, Jiayu Li, Ian Mertz, and Edward Pyne. The structure of catalytic space: Capturing randomness and time via compression. *Electronic Colloquium on Computational Complexity: ECCC*, 2024.

- 6 Dean Doron, Edward Pyne, and Roei Tell. Opening up the distinguisher: A hardness to randomness approach for $BPL=L$ that uses properties of BPL. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024*, pages 2039–2049, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649772.
- 7 Michal Koucký, Ian Mertz, Edward Pyne, and Sasha Sami. Collapsing catalytic classes. *Electron. Colloquium Comput. Complex.*, TR25-019, 2025. URL: <https://eccc.weizmann.ac.il/report/2025/019>.
- 8 Ian Mertz. Reusing space: Techniques and open problems. *Bulletin of EATCS*, 141(3), 2023. URL: <http://eatcs.org/beatcs/index.php/beatcs/article/view/780>.
- 9 Noam Nisan. On read once vs. multiple access to randomness in logspace. *Theoretical Computer Science*, 107(1):135–144, 1993. doi:10.1016/0304-3975(93)90258-U.
- 10 Edward Pyne. Derandomizing Logspace with a Small Shared Hard Drive. In Rahul Santhanam, editor, *39th Computational Complexity Conference (CCC 2024)*, volume 300 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CCC.2024.4.
- 11 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.