Frontier Space-Time Algorithms Using Only Full Memory

Petr Chmel
Computer Science Institute of Charles University, Prague, Czech Republic

Aditi Dudeja
The School of Data Science, The Chinese University of Hong Kong, Shenzhen, China

Michal Koucký
Computer Science Institute of Charles University, Prague, Czech Republic

Ian Mertz
Computer Science Institute of Charles University, Prague, Czech Republic

Ninad Rajgopal
Computer Science Institute of Charles University, Prague, Czech Republic

Abstract

We develop catalytic algorithms for fundamental problems in algorithm design that run in polynomial time, use only $\mathcal{O}(\log(n))$ workspace, and use sublinear catalytic space matching the best-known space bounds of non-catalytic algorithms running in polynomial time.

First, we design a polynomial time algorithm for directed s-t connectivity using $n/2^{\Theta(\sqrt{\log n})}$ catalytic space, which matches the state-of-the-art time-space bounds in the non-catalytic setting [20], and improves the catalytic space usage of the best known algorithm [43]. Furthermore, using only $\mathcal{O}(\log(n))$ random bits we get a randomized algorithm whose running time nearly matches the fastest time bounds known for space-unrestricted algorithms.

Second, we design polynomial time algorithms for the problems of computing Edit Distance, Longest Common Subsequence, and the Discrete Fréchet Distance, again using $n/2^{\Theta(\sqrt{\log n})}$ catalytic space. This again matches non-catalytic time-space frontier for Edit Distance and Least Common Subsequence [58].

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Complexity classes

Keywords and phrases Catalytic computation, Connectivity, Edit distance, Discrete Fréchet distance

Digital Object Identifier 10.4230/LIPIcs.CCC.2026.29

Related Version *Full Version*: https://arxiv.org/abs/2602.21089

Funding *Michal Koucký, Ian Mertz, Ninad Rajgopal:* Partially supported by the Grant Agency of the Czech Republic under the grant agreement no. 24-10306S and by Charles Univ. project UNCE 24/SCI/008.
Petr Chmel: Partially supported by the Grant Agency of the Czech Republic under the grant agreement no. 24-10306S, by GAUK project No. 70924 and by Charles Univ. project UNCE 24/SCI/008.
Aditi Dudeja: This work was initiated while the author was affiliated with the University of Salzburg. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 947702).

1 Introduction

1.1 Catalytic Computation

Recently, a new paradigm for space-bounded computation called *catalytic computing* has received a great deal of attention. In short, a catalytic machine is one whose memory is almost completely full with arbitrary data at the start; while it may freely use this full memory, known as *catalytic space*, in a read-write manner as with ordinary space, it must restore the initial data to the catalytic memory at the end of the computation. Catalytic computing was defined by Buhrman, Cleve, Koucký, Loff, and Speelman, in [32], who showed that the addition of polynomial catalytic space to a log-space machine gives it significant power, more so even than non-determinism, randomness, or both.

Since then there have been a host of results regarding the catalytic model, such as structural results [34, 51, 44, 41, 39, 60], placing further problems into the model [63, 6, 8, 5], and alternative catalytic models [23, 52, 50, 33, 24, 38] (see surveys of Koucký [59] and Mertz [62] for more background). Furthermore, the techniques of using full memory were also used in the context of ordinary space-bounded computation, giving rise to a novel paradigm for derandomization [45, 61, 46, 53] as well as shedding new light on the relationship of space and time [40, 42, 70, 68].

While all problems in NL are computable in polynomial time CL (i.e., the class of problems computable by a log-space machine with polynomial catalytic space), the approach of [32] incurs a large overhead in both time and catalytic space. In response to this, there been a recent focus, responding to a question posed by Mertz [62], of truly efficient catalytic algorithms whose time and catalytic space usage are both comparable to the best known results in the non-catalytic setting. Cook and Pyne [43] showed that connectivity in graphs (STCONN) can be solved in quadratic time using linear catalytic space; they also show a similar result for estimating random walks. These algorithms are conceptually simple enough to be implemented in practice – to wit, they are only a $\mathsf{poly}(\log(n))$ factor slower than breadth-first search – and redevelop pre-existing techniques from the catalytic literature in new and streamlined ways.

In this work we push the algorithmic frontier of connectivity and other problems, by asking whether we can capture the best *space-time* algorithms, namely those using (reasonable) polynomial time and sublinear space, entirely within the catalytic paradigm.

1.2 Sublinear-Space Polynomial-Time Algorithms

Connectivity is an extremely fundamental primitive in graph algorithms, both in theory and in practice. In complexity theory the many variants of STCONN characterize logspace and their adjacent classes, while on the algorithmic side there has been an extensive history of studying its space, time, space-time trade-offs and heuristics.

While the straightforward DFS and BFS algorithms for STCONN use linear time in the number of edges, the space requirement for these algorithms is linear in the number of vertices. In contrast, Savitch's theorem shows that this problem can be solved in $\mathcal{O}(\log^2(n))$ space and time $n^{\mathcal{O}(\log(n))}$ [67]. To meet both conditions, i.e. run in polynomial time and use sublinear space, a result by Barnes, Buss, Ruzzo, and Schieber [20] gives space complexity $\frac{n}{2^{\Theta\left(\sqrt{\log(n)}\right)}}$.

Except for restricted graph classes [10, 55, 65, 64], no better results are known. Furthermore, the algorithm of [20] can be implemented in a restricted model of computation called Node Naming Jumping Automata on Graphs (NNJAG), for which we have a matching space lower bound in the polynomial time regime due to Edmonds, Poon, and Achlioptas [48], even for probabilistic algorithms.

There are also a number of fundamental algorithmic questions which reduce to connectivity, including Edit Distance (ED), Longest Common Subsequence (LCS), and Discrete Fréchet Distance (DFD). The first two, which quantify how close two strings are in various ways, are very well-studied problems with applications in various areas such as data analysis, computational biology, and text processing [25, 1, 16, 17, 18, 2, 3, 11, 12, 13, 14, 15, 19, 21, 26, 29, 27, 28]. Meanwhile, DFD has been studied extensively in computational geometry, motivated by applications such as dynamic time-warping [56], matching of time series in databases [57], and analysis of moving objects [30, 31].

For ED and LCS, the classic dynamic programming algorithms need linear space, and similarly for DFD, the original dynamic programming algorithm of Eiter and Mannila [49] takes $\mathcal{O}(mn)$ time (where m and n are numbers of vertices of the polygonal chains); this was subsequently improved to $\mathcal{O}(n)$ space and $\mathcal{O}\left(\frac{mn\log\log(n)}{\log(n)}\right)$ time by Agarwal et al. [4]. On the other hand, if we allow for $n^{\mathcal{O}(\log(n))}$ runtime, then we can solve these problems in $\mathcal{O}(\log^2 n)$ space via a reduction to STCONN. With regards to space-time results, their respective reductions to STCONN involve a quadratic blowup in the input size n, which would render the result of [20] useless; however, a recent work of Kiyomi, Horiyama, and Otachi [58] gives a polynomial time algorithm for LCS and ED using $\frac{n}{2^{\Theta\left(\sqrt{\log(n)}\right)}}$ space. While better algorithms are known in the approximation and/or streaming regimes [35, 16, 66, 37], any improvements for these problems beyond the STCONN threshold seem firmly out of reach [37].

1.3 Our Results

In this work, we develop polynomial-time algorithms using catalytic space that matches the $\frac{n}{2^{\Omega\left(\sqrt{\log(n)}\right)}}$ space-bound frontier for polynomial-time algorithms that solve the aforementioned problems, with all the space being borrowed from full memory except for $\mathcal{O}(\log n)$ bits.

We begin with STCONN, where additionally $\mathcal{O}(\log(n))$ random bits are sufficient to nearly match the time bounds of the best space-*unrestricted* algorithms.

▶ **Theorem 1** (Sublinear catalytic space algorithms for STCONN). *There exists a deterministic algorithm solving* STCONN *on n-vertex graphs in time* poly(n)*, space $\mathcal{O}(\log(n))$, and catalytic space $\frac{n}{2^{\Omega\left(\sqrt{\log(n)}\right)}}$.*

Furthermore, for every $\epsilon > 0$, there is a randomized algorithm that runs in time $\mathcal{O}(|E| \cdot n^{2+\epsilon})$ given $\mathcal{O}(\log(n))$ random bits which uses the same space (both ordinary and catalytic) as the deterministic algorithm.

The randomized algorithm only makes *one-sided errors*, where it always answers correctly when s is not connected to t in G. As such, the deterministic algorithm stated above is obtained by direct derandomization within the same space, i.e., by going over all possible random strings and accepting, if at least one of them makes the randomized algorithm accept. Our algorithms also improve on the $\tilde{O}(n)$ (or the $\tilde{O}(n^2)$) catalytic space bound observed by the randomized (or deterministic) algorithms from [43], achieving a sublinear bound in both cases.

Next, we extend this result to ED, LCS, and DFD, by designing catalytic algorithms that exploit the special graph structure underlying the problems. To the best of our knowledge, no sublinear space algorithm was known for DFD even without the catalytic restriction.

▶ **Theorem 2** (Sub-linear catalytic space algorithms for ED, LCS, and DFD)**.** *There exist deterministic algorithms for* ED*,* LCS*, and* DFD *which use time* $\mathsf{poly}(n)$*, space* $\mathcal{O}(\log(n))$*, and catalytic space* $\frac{n}{2^{\Omega\left(\sqrt{\log(n)}\right)}}$*.*

Furthermore, the running time of our algorithm for DFD *is* $\mathcal{O}(n^{4+\varepsilon})$*, for any* $\epsilon > 0$*, if all the distances of the points are integers from the range* $\{0, \ldots, \mathsf{poly}(n)\}$*.*

We do not know what is the exact running time of our algorithms for edit distance and longest common subsequence as it depends on the running time of a log-space procedure[1] for conversion between Chinese remainder representation and the usual binary representation of integers [54].

To the best of our knowledge, no sublinear space algorithm was known for DFD even without the catalytic restriction. However, we note that such a theorem can be derived via the reduction to STCONN and existing results: Chakraborty and Tewari [36] give a polynomial time algorithm for STCONN in directed layered planar graphs requiring $\mathcal{O}(n^{\varepsilon})$ space which we can directly plug in to obtain a deterministic algorithm for DFD that uses time $\mathsf{poly}(n)$ and space $\mathcal{O}(n^{\varepsilon})$.

This result is incomparable to our catalytic algorithm, as it uses less space overall but significantly more free memory. We also note that the results on grid-graph reachability of Allender et al. [9] are not sufficient for our purposes as in the process of reducing DFD to grid reachability, some instances can have both multiple sources and multiple sinks. Moreover, the grids we consider are NL-complete as opposed to the planar grids considered in [9] which are not known to be NL-complete.

Independently, Edenhofer [47] considered catalytic space-work space tradeoffs for STCONN using techniques similar to ours: for any $k \in [n]$, he gives an algorithm for solving STCONN using $\mathcal{O}(\log n \cdot \log k + \log n)$ space and $\mathcal{O}\left(\frac{n}{k} \cdot \log^2(n)\right)$ catalytic space.

1.4 Proof Overview

Our technique will primarily rely on the structure of two different STCONN algorithms: first, the sublinear-space polynomial-time algorithm of Barnes et al. [20] in the non-catalytic world; and second, the recent polynomial-time algorithm of Cook and Pyne [43] using linear catalytic space. We merge their approaches using analysis from other areas of catalytic computing, as well as incorporating new graph tools for overcoming one of the core barriers that arises in the approach of [20], to resolve the case of STCONN. Lastly, we use a common framework to attack ED, LCS, and DFD, working on an implicit grid graph whose structure will allow us to focus on small subgraphs and thus keep our space small.

1.4.1 Previous algorithms for STCONN: [20] & [43]

1.4.1.1 Long and Short Paths Framework ([20])

The starting point for Theorem 1 is the deterministic algorithm by [20]. For any parameter λ, [20] constructs a *representative set* of vertices $U \subset V$ of size roughly n/λ such that, if any pair u is connected to v in G by a path $\rho_{u,v} = (u, p_1, \ldots, p_L, v)$, then there exists a sequence of vertices $\tilde{\rho}_{u,v} = (u, w_1, \ldots, w_{L'}, v)$, such that each w_i is in U, and consecutive vertices in $\tilde{\rho}_{u,v}$ are connected by a path of length at most λ. This provides a sparsification $H = (U, E_U)$

[1] The running time for this procedure was not explicitly calculated and it boils down to the running time of a highly sophisticated log-space algorithm for multiplying n integers of $\mathcal{O}(\log n)$ bits each.

of G that still preserves connectivity within G: (s_i, s_j) is in E_U, if and only if there exists a path of length λ from s_i to s_j in G. Thus by setting λ to be $2^{\mathcal{O}\left(\sqrt{\log(n)}\right)}$, storing U in the workspace suffices to decide STCONN efficiently, if one can, in polynomial time and n/λ workspace,

- **construct** the representative set U;
- check if (s_i, s_j) forms an edge in the sparsified graph H, through a "**short path**" algorithm that checks for connectivity over paths of length λ in G; and
- using this short-paths algorithm as a sub-routine, find a "**long path**" in H, say $w_1, \ldots, w_{L'}$, such that the distances from s to w_1, and from $w_{L'}$ to t are at most λ in G.

To construct the set U, [20] employs a breadth-first search (BFS) from s to partition the vertices into λ candidate sets $U_1, \ldots, U_\lambda$, where intuitively each U_i contains the vertices at distance $i, i+\lambda, \ldots, i+\lfloor n/\lambda \rfloor \cdot \lambda$ from s in the BFS-tree. Each U_i is a representative set by design, and by a simple averaging argument one of these sets, which we choose to be our true representative set, has at most n/λ vertices. Thus if our "short paths" algorithm can test whether two nodes are at distance exactly λ, the vertices and edges of our sparsified graph H can be computed in polynomial time and $|U| = \mathcal{O}(n/\lambda)$ space using BFS; this gives us our "long paths" algorithm.

To check if vertices u and v are connected by a path of length λ in G, [20] interpolate between ordinary BFS and the recursive approach by Savitch [67]. As a first step to reduce the space, they arbitrarily partition set V into λ color classes, each of size n/λ, and look only for u–v paths which traverse over a fixed sequence of $\lambda + 1$ (possibly repeatable) color classes, where the first class contains u and the last class contains v. This is achieved by a basic BFS algorithm, iteratively computing connectivity between consecutive color classes in the sequence edge-by-edge and only storing the bit-vector of length n/λ of which nodes in the current color class are reachable from u.

By looping over all possible sequences of color classes this discovers every possible path of length λ; however, the runtime of this procedure is at least λ^λ, which is superpolynomial when λ is $2^{\sqrt{\log(n)}}$ as desired. Taking inspiration from Savitch's recursive algorithm, they instead choose a middle color class and test connectivity from the first color class, i.e. the one containing u, to this middle class and then from this middle class to the final class containing v. Such a test reduces connectivity of distance λ to 2λ tests of distance $\lambda/2$, and recursively this gives a runtime of $(2\lambda)^{\log \lambda}$ instead, giving us our final choice of $\lambda = 2^{\mathcal{O}(\sqrt{\log(n)})}$.

We aim to design a polynomial time algorithm for STCONN which uses only $\mathcal{O}(\log(n))$ workspace and $\mathcal{O}(n/\lambda)$ catalytic space. The first issue with adapting their approach is that storing U requires $\mathcal{O}(n/\lambda)$ clean workspace, which we cannot afford. Furthermore, even if one generates U in low-space, it is unclear how to construct catalytic algorithms for computing long and short paths that satisfy our time-space requirements. With regards to the second issue, we now turn to the catalytic algorithm of Cook and Pyne [43], using $\mathcal{O}(\log(n))$ workspace and $\tilde{\mathcal{O}}(n)$ catalytic space, which gives us an initial clue for our subroutines.

Connectivity via Propagating Sums ([43])

In a word, the Cook-Pyne algorithm implements a catalytic version of BFS. To illustrate their main ideas, consider a layered graph G with n layers of n vertices each, with edges only going between consecutive layers, and we want to test connectivity between s in the first layer and t in the last. We describe an algorithm using $\tilde{\mathcal{O}}(n^2)$ catalytic space, which is linear in the number of vertices of G (of course their algorithm works on arbitrary graphs, albeit with slightly less efficiency than in the layered case).

Let $R \coloneqq \mathbb{Z}/2^m\mathbb{Z}$ be the ring of integers modulo 2^m, for some m which we fix shortly. Their algorithm first allocates n^2 registers $r_{1,1} \dots r_{n,n}$ on the catalytic tape, with each $r_{i,j}$ being assigned to corresponding node $v_{i,j}$ in G, and let their initial contents be $\tau_{1,1} \dots \tau_{n,n} \in R$; this gives a total catalytic memory of $n^2 \cdot m$ bits. They start by "pushing" the value of $r_{1,j}$ through all its outgoing edges to the second layer, in particular by adding $\tau_{1,j}$ to each node $v_{2,j'}$ connected to $v_{1,j}$. They then push the values of the $r_{2,j}$ nodes forward to the third level, where now each $r_{2,j}$ has not just $\tau_{2,j}$ but also any value added to it in the previous round.

Repeating this procedure for all layers i in turn, we obtain some final value σ which arrives at the vertex (n, t) at the end of this process. They reset all catalytic memory by performing a "reverse edge push", i.e. repeating the whole procedure but with subtraction and in reverse, thus getting back $\tau_{i,j}$ in each register $r_{i,j}$. Finally, they repeat the entire algorithm but with the register $r_{1,s}$ for the source incremented by 1, collecting the value σ' at the vertex (n, t) and resetting as before.

For each pair of nodes $(v_{1,i}, v_{n,j})$ in the first and final layer respectively, let $\Gamma_{i,j}$ be the collection of paths between them. By linearity, the net result of our first forward push before resetting is

$$r_{n,j} \mathrel{+}= \sum_{i \in [n]} \sum_{P \in \Gamma_{i,j}} \sum_{v_{k,\ell} \in P} \tau_{k,\ell} \pmod{2^m}$$

because each value $\tau_{k,\ell}$ is added forward along the path P. In particular this holds for t, which gives us our σ value, while the same argument in the second forward push gives

$$\sigma' = \left(\sum_{i \neq s} \sum_{P \in \Gamma_{i,t}} \sum_{v_{k,\ell} \in P} \tau_{k,\ell} \right) + \left(\sum_{P \in \Gamma_{s,t}} \left(1 + \sum_{v_{k,\ell} \in P} \tau_{k,\ell} \right) \right) \pmod{2^m}$$

Hence $\sigma' - \sigma$ is exactly $|\Gamma_{s,t}| \pmod{2^m}$, i.e. the number of paths from s to t modulo 2^m.

When $2^m > \sigma' - \sigma$ this resolves the connectivity between s and t and even accurately counts the exact number of s–t paths. However, the number of paths can be as large as $\mathcal{O}(n^n)$, and setting each register size to be of size $m = \tilde{\mathcal{O}}(n)$ increases the catalytic space as well as the run-time by a factor of at least n. To deal with this, they finish the argument by using the fact that with large probability, a random prime p of value $\mathsf{poly}(n)$ satisfies $\sigma' - \sigma \not\equiv 0 \pmod{p}$ if the number of paths is non-zero. Hence, instead of computing over the ring R they compute over the field $\mathbb{F}_p$, thus reducing the catalytic space to $\mathcal{O}(n^2 \cdot \log n)$ as required.

Assuming one could stitch shorter segments together, we could use this algorithm for the long path procedure (putting aside the layered issue) on the graph H of representative nodes. However, using this for our short-paths sub-routine would still require the algorithm to push values across the edges associated with all the n vertices, necessitating catalytic space of $\tilde{\mathcal{O}}(n)$.

1.4.2 Our approach for STCONN

We now move on to our approach. The two conceptual contributions of our work are the following, which we discuss in turn: 1) we adapt the flow-pushing argument of [43] to the recursive framework of [20] to get the best of both worlds; and 2) we construct the representative set U for the long paths algorithm using tools from the theory of pseudo-randomness in a space and time efficient manner.

Merging Recursion and Flows

Assuming we have access to our representative set U, the long paths procedure can be handled by the [43] algorithm for unlayered graphs, and so we focus on finding paths of length λ in G. In combining the flow-based algorithm of [43] with the recursive approach of [20], we show both that the catalytic space of [43] can be reduced using techniques from [20] and that the space of [20] can be made catalytic using [43].

If we want to adopt the recursive structure of [20], then we cannot use the idea of adding 1 to the initial vertex and comparing the sums before and after; this would require too much free memory and would make it unclear how to "stitch together" the two sides of a recursive call. Instead, we take inspiration from previous catalytic algorithms, such as that of Ben-Or and Cleve [22], which handle arithmetic cancellations in recursion. Our goal will be to count the number of paths between any pair of vertices u and v.

For $\ell = 1 \dots \log(\lambda)$, let $\Gamma^{\ell}_{u,v}$ be the set of u–v paths of distance exactly 2^{ℓ}. [22] give the following recursive invariant:

$$r_v \mathrel{+}= \tau_u \cdot |\Gamma^{\ell}_{u,v}|$$

Following [20], we may consider all possible midpoints for a u–v path of length 2^{ℓ}. Where before we simply took $\vee_w (u \to_{2^{\ell-1}} w) \wedge (w \to_{2^{\ell-1}} v)$, the crucial insight is that moving to arithmetic $\vee$ and $\wedge$, i.e. $+$ and $\times$, faithfully counts the number of paths in the same way, as any u–w path can be paired with any w–v path to give a u–v path:

$$|\Gamma^{\ell}_{u,v}| = \sum_w |\Gamma^{\ell-1}_{u,w}| \cdot |\Gamma^{\ell-1}_{w,v}|$$

Thus utilizing the [22] procedure for recursive multiplication, fixing a midpoint w, allows us to evaluate this inner product using four recursive calls:

1. $r_v \mathrel{-}= r_w \cdot |\Gamma^{\ell-1}_{w,v}|$ // $r_v = \tau_v - \tau_w \cdot |\Gamma^{\ell-1}_{w,v}|$
2. $r_w \mathrel{+}= r_u \cdot |\Gamma^{\ell-1}_{u,w}|$ // $r_w = \tau_w + \tau_u \cdot |\Gamma^{\ell-1}_{u,w}|$
3. $r_v \mathrel{+}= r_w \cdot |\Gamma^{\ell-1}_{w,v}|$ // $r_v = \tau_v + \tau_u \cdot |\Gamma^{\ell-1}_{u,w}| \cdot |\Gamma^{\ell-1}_{w,v}|$
4. $r_w \mathrel{-}= r_u \cdot |\Gamma^{\ell-1}_{u,w}|$ // $r_w = \tau_w$ (and $r_u = \tau_u$)

Repeating this for all potential midpoints w gives us the outer sum, and hence the invariant above. Repeating for every u–v pair, by linearity we arrive at a final value of

$$r_v \mathrel{+}= \sum_u \tau_u \cdot |\Gamma^{\ell}_{u,v}| \quad \forall v \in V$$

which we take as the recursive invariant for our algorithm. This fits the base case of $\ell = 0$, since we add r_u to r_v for each u which has an edge to v, while the analysis of [22] shows that it holds for ℓ given access to $\ell - 1$. Crucially for the catalytic restriction, at the end of each subroutine and hence the entire algorithm, r_u and r_w are left in their original configuration.

We now turn to the time of the recursion. The above procedure, looping over all w independently, is the structure of Savitch's algorithm [67], and so following [20] we generalize our recursive calls to work over choosing middle color classes rather than individual vertices. Once again we choose to have λ color classes of size n/λ each, and in making four recursive calls per middle color class (rather than the two from [20]), our runtime becomes $(4\lambda)^{\log(\lambda)}$, which does not change our asymptotics for $\lambda = 2^{\sqrt{\log(n)}}$. Our vectors of length n/λ are entirely catalytic, with our free space being only the $\mathcal{O}(\log n)$ necessary to compute sums and keep track of our runtime.

As a final note, to make the analysis of our recursive condition simpler, we use the *same* recursive algorithm for the long paths as well, with some minor interfacing between the two to reflect the change in graphs. In the outer algorithm we run for paths of length up to $|U|$,

i.e. maximal length paths, but in exchange we remove the color class structure and operate on the entire graph at every step. This gives a runtime of $4^{\log |U|} = \mathcal{O}(n^2)$, which is linearly less efficient than [20] and [43].

Efficiently Constructible Representative Sets

We return to our final loose end for STCONN, which is the construction of the representative set for our long paths algorithm using low space. To do this, we use *pairwise-independent hash function families* $\mathcal{H}_{m,n}$ to construct a family of *multisets* $\{U_\sigma\}$, indexed by $\sigma \in \{0,1\}^{(6+\varepsilon)\log(n)}$, and prove that with high probability over the choice of σ, a multi-set from this family is a representative set for G. For convenience of exposition, we assume $\{U_\sigma\}$ as a family of sets, and deal with the technical challenges that arise in the STCONN algorithm from U_σ being a multi-set in Section 4.

Recall (from [69]) that for $m \leq n$, $\mathcal{H}_{m,n}$ is a pair-wise independent family of hash functions $h : [m] \to [n]$ indexed by $2\log(n)$ bits, such that an h chosen at random from $\mathcal{H}_{m,n}$ maps any pair $i \neq j \in [m]$ into $[n]$ independently. Moreover, each hash function can be evaluated on an input $i \in [m]$, in $\mathsf{poly}(\log(n))$ time and $\mathcal{O}(\log(n))$ space (see Lemma 10 for the formal statement).

For the construction, we set $m = \mathcal{O}(n/\lambda)$, and suppose that we sample $K = \mathcal{O}(\log(n))$ many hash functions $h_1, \ldots, h_K$ from $\mathcal{H}_{m,n}$, and consider U_σ as the union of their images, where σ is the random seed used to sample them. Using a probabilistic argument, we show that with high probability over the choice of the $h_1, \ldots, h_K$ (or σ) the following holds: for every pair of vertices u, v from G that are connected by a path of length $\lambda/2$, there exists a walk between them traversing $\lambda/2 - 1$ distinct vertices that visits U_σ.

With high probability over σ, U_σ forms a representative set for G, over paths of length λ. Indeed, for any u, v that is connected, suppose we break this path into L' sub-paths of size $\lambda/2$. By the properties of a "good" U_σ, we see that each sub-path can be replaced by a walk traversing $\lambda/2-1$ distinct vertices and visiting U_σ. The distance between consecutive vertices from U_σ along this new path from u to v is at most λ, so U_σ can be used to sparsify G.

However, the length of σ is $2K\log(n) = \mathcal{O}(\log^2(n))$, and we cannot store this within our $\mathcal{O}(\log(n))$-sized workspace. As such, we use the well-known technique of reducing randomness by performing a random walk on a *constant-degree explicit expander* D, with suitably high expansion [7]. In particular, sampling K hash functions using a random walk on D specified over vertex set $\{0,1\}^{2\log(n)}$ (that describes all of $\mathcal{H}_{m,n}$), gives similar guarantees of intersecting with walks, as that of picking K hash functions uniformly at random. The number of random bits is now at most $\mathcal{O}(\log(n))$, and by carefully accounting the values of K, the degree and expansion of D, a seed length of $(6+\varepsilon)\log(n)$ suffices to describe U_σ.

Finally, any vertex in U_σ can be generated, given σ, in $\mathsf{poly}(\log(n))$ time and $\mathcal{O}(\log(n))$ space. This holds from standard explicit constructions of constant-degree expanders, and efficient evaluation of hash functions in $\mathcal{H}_{m,n}$.

1.4.3 ED, LCS, & DFD Without Direct Reductions To STCONN

We now turn our attention to computing ED, LCS, & DFD. A natural direction would be to use the reduction to reachability of grid-like graphs and use the previous algorithm. However, on inputs of length n, the underlying grid-like graph has quadratic size, and thus the space required would be $\frac{n^2}{2^{\Omega(\sqrt{\log n})}}$ – far larger than the space we would like to use.

LCS and ED in Sublinear Free Space ([58])

While [20] have the outer algorithm with a single color class (corresponding to the representative set) and recursion depth $\mathcal{O}(\log n)$ and the inner algorithm with λ color classes and recursion depth $\mathcal{O}(\log(\lambda)) = \mathcal{O}(\sqrt{\log n})$, [58] approach the problem by effectively using the ideas similar to the inner algorithm of [20] as their outer algorithm. In the outer algorithm, given a color class in the first layer[2] and a color class in the last layer, [58] choose a middle layer of the grid-like graph and then iterate over the λ color classes of size n/λ as the possible choices of a middle vertex of the path, doing this recursively until a base case is reached, when the two layers are at distance n/λ. In this base case, the inner algorithm takes over. Because the grid-like graph only has edges to close neighbors, either the two color classes are far enough so that there is no possible path between the two, or the graph containing the two color classes and all paths between them contains $\mathcal{O}(n/\lambda)$ rows and $\mathcal{O}(n/\lambda)$ columns, and therefore we can compute the shortest path from the first color class to the second color class by the standard dynamic programming algorithm that only requires us to remember two rows at a time.

It is not immediate that one can emulate the algorithm of [58] using catalytic tools. In particular, their algorithm is designed to find the shortest path which is necessary for computing edit distance. Thus the algorithm always has to choose the shortest path that leads to a given middle point. However, the "push" algorithm of Cook and Pyne sums all the paths leading to the middle point. It is not clear how to select the minimum path among them. More to the point, the values we are summing in the algorithm of Cook and Pyne have nothing to do with the true values we want to take minimum of as they are offset by some unknown initial values from the catalytic memory. Taking their minimum would be completely useless.

The standard approach to overcome this would be to layer the graph and count the length of the paths implicitly. Checking whether the length of a path from a source vertex to the target vertex is of length at most d is done by checking whether the source vertex in the first layer is connected to the target vertex in the d-th layer. However, the layering increases the size of the graph by a factor of $\mathcal{O}(n)$ so any possible savings in space from clever algorithms will be wiped-out.

To overcome this, we stick to the original graph but introduce multiplicative weights to the edges. It turns out the the algebraic framework of the Cook-Pyne algorithm can be modified to count a *weighted sum* of the paths. The weight of a path will be the product of its edge weights. By choosing weights that are large enough, different path weights will be separated so that from their weighted sum we can read off the count of the paths of different weights. This assumes we will work with large integers that we will not have space to store; the path weights will have $\mathcal{O}(n^2)$ bits. We overcome this by computing the final value modulo different primes, that is we compute it in Chinese remainder representation. Using a space-efficient algorithm that converts Chinese remainder representation into the usual binary representation, we get bit-by-bit access to the actual weighted sum. From it we can read off the length of the shortest path as well as the length of the longest path.

So our goal will be to design an algorithm for weighted grid graphs that computes the weighted sum of paths between two vertices modulo a small prime.

[2] The grid-like graph is not layered in the usual sense, which is not an issue in this algorithm as it only uses free space.

Weighted Reachability on Grid Graphs

We use a similar high-level technique to finding the shortest path in grid-like graphs as the algorithm [58]. Instead of finding the shortest path we will compute the weighted sum of the paths.

Again, our algorithm for weighted reachability on grid graphs will be composed of an outer and an inner algorithm. For practical purposes, we assume that the edge weights are given to us by an oracle, and we use this oracle as a parameter for our three functions of interest which we want to compute. Moreover, for this algorithm we assume all values to be over a field $\mathbb{F} = \mathbb{F}_p$ for some prime $p = \mathsf{poly}(n)$.

The outer algorithm will be essentially identical to the inner "short paths" algorithm from our algorithm for general STCONN. It will use recursion of depth $\mathcal{O}(\sqrt{\log n})$ with color classes of size n/λ, for $\lambda = 2^{\Theta(\sqrt{\log n})}$, to reduce the weighted reachability on a grid of size $n \times n$ to the weighted reachability on grid-graphs of size roughly $n/\lambda \times n/\lambda$. This smaller reachability will be solved by the inner algorithm.

However, we do not have the luxury of using our previous algorithm as our inner algorithm, given that our goal is to use only $\mathcal{O}(\log n)$ free space. Therefore, we use a similar idea to the outer algorithm of [20] together with the ideas of Cook and Pyne. We are interested in computing weighted reachability for two color classes of size n/λ at horizontal distance n/λ. Two color classes at such horizontal distance in a grid-graph can affect each other only if they are adjacent vertically. Thus, we get a subgrid with n/λ layers and at most three color classes, yielding $3 \cdot n/\lambda$ vertices per layer. We now use the idea of propagating sums for the outer algorithm in our STCONN algorithm while recursing again on the middle layers. We will now have only a single color class, which is not an issue with $\mathcal{O}(n/\lambda)$ vertices per layer. Our final base case here is the case of two neighboring layers, where we can iterate over all $\mathcal{O}(n/\lambda)$ edges and multiply intermediate values by their weight.

Here, we only need a constant number of bits of free space per level of recursion for the instruction counter, and therefore the total recursion depth $\mathcal{O}(\log n)$ does not pose any large issues space-wise. Moreover, we can implement the recursion in a constant number of steps per each recursion level, thus we get time complexity $c^{\mathcal{O}(\log n)} \cdot \mathcal{O}(n/\lambda) = \mathsf{poly}(n)$. Regarding catalytic space usage, if we use a new catalytic register vector at every recursion level, we require $\mathcal{O}(\log n)$ vectors, each using n/λ field elements, which each requires $\mathcal{O}(\log p)$ bits, which in total yields the required catalytic space complexity $\frac{n}{2^{\Omega(\sqrt{\log n})}}$.

To finally compute the weight of all u, v-paths modulo p, we first run the algorithm forward, recording the result r_1, then rerun the computation backwards. We than add 1 to the register corresponding to u, and run the computation again, recording the result r_2, and rerun the computation backwards for the final time. We then subtract 1 from the register corresponding to u and return $r_2 - r_1$ as the result.

Using Grid Reachability to Compute Discrete Fréchet Distance

For discrete Fréchet distance of point sequences $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_n)$, we use the previous algorithm as a blackbox, and we provide a suitable edge weight oracle. We also assume that the distances of the points are integers in the range $\{0, \ldots, \mathsf{poly}(n)\}$. If we are given a more complicated metric, we can build a range query oracle out of it – that is, we can compute how many different point pairs have distance lower than the queried point pair (p, q), by iterating over all the other pairs, and thus we would get an oracle whose values are in the range $\{1, \ldots, n^2\}$.

One possible way of computing discrete Fréchet distance is by taking a grid-like graph on vertex set $\{0,\dots,n\}\times\{0,\dots,n\}$, where each vertex (i,j) has three outgoing edges into vertices $(i+1,j),(i,j+1),(i+1,j+1)$ if these vertices exist. We then build an induced subgraph which we call $\mathsf{Frech}_{P,Q,m}$, where we set the weight of each edge $((i,j),(k,\ell))$ to one if the distance between p_k and q_ℓ is at most m and zero otherwise. As there are only $\mathsf{poly}(n)$ possible values for m, we can try all possible values of m, and the smallest value of m such that there is a path between $(0,0)$ and (n,n) is the Fréchet distance. However, there may be exponentially many paths and we only allow computation modulo primes of polynomial size. We remedy this using the Chinese remainder theorem and the fact that the total weight is zero iff the total weight module the first $4n$ primes is zero. Thus, we run the algorithm for $4n$ primes with the same weights (as they are all either zero or one) to determine whether the discrete Fréchet distance is at most m, and we run binary search over all choices of m to compute DFD in polynomial time, logarithmic free space and catalytic space $\frac{n}{2^{\Omega(\sqrt{\log n})}}$.

Using Grid Reachability to Compute ED and LCS

For ED and LCS for n-character strings, we want to find the shortest or, respectively, longest path in a weighted grid-like graph – essentially the graph for DFD. However, unlike DFD, we need to take care of different weights of edges. Furthermore, in our approach, we multiply weights instead of adding them, and hence we must move to addition in the exponent.

For the grid-like graph, we note that there at most 3^{2n} paths (the longest path has length $2n$, and each vertex has out-degree at most three). In all cases, the edges in the grid-like graph have two possible weights: 0 or 1, but some edges might also be missing. Therefore, we can simulate addition by having the weights as follows: if the edge is missing, its weight will be set to 0. (And therefore, any path using such edge will have zero total weight, which effectively removes the edge.) For the existing edges, if an edge had weight $\mu(e)$ originally, then our new weight will be $16^{n\cdot\mu(e)}$. Therefore, weight-zero edges will have their new multiplicative weight set to 1, and weight-one edges will have their multiplicative weight set to 16^n. This also ensures that we can efficiently learn the longest/shortest path in the graph: given the result, which is a sum of weights of all s,t-paths, the lowest bit set to 1 corresponds to the length of the shortest path: indexing bits such that the least significant bit has index 0, any of the bits $4n(w+1)-1,\dots,4nw$ being set to one corresponds to there existing a path of weight w in the original additive weighting scheme. Therefore, we can just check all the bits of the number to find the largest and the smallest non-zero bit index.

There is one slight issue: if we were computing this over integers, the representation of the value r would have $\Theta(n^2)$ bits, which we cannot afford to represent directly. However, we can use the Hesse-Allender-Barrington algorithm [54] for reconstructing integers from their Chinese remainder representation that uses polynomial time, polynomial number of queries for the value of the r mod p_i for some primes p_i and logarithmic space to output a particular bit of r in the binary representation. Since our algorithm returns all results modulo p, each query to r mod p_i corresponds to us running our reachability algorithm over the field $\mathbb{F}_{p_i}$. Thus, we get again polynomial time, logarithmic free space and catalytic space $\frac{n}{2^{\Omega(\sqrt{\log n})}}$.

We note that this approach also works for the weighted versions of edit distance if the weights are $\mathcal{O}(\log n)$-bit integers with appropriate changes to the weighting scheme.

1.5 Future Work

We consider these algorithms to be an initial step towards bypassing the current $\frac{n}{2^{\Theta\left(\sqrt{\log(n)}\right)}}$ space barrier for these problems. The NNJAG lower bounds which substantiate this barrier are based on pebbling techniques, which was also the case for Tree Evaluation for over

a decade before finally being broken by the catalytic algorithm of [42]. Their algorithm bypasses these pebbling lower bounds by combining multiple pebbles in the same location, a process which has been called "fuzzy pebbling" by later reviews on the topic. This is also the situation for our STCONN algorithm, where the connectivity of many nodes in the graph are all stored on top of one another; thus there is hope that further development of catalytic techniques or more clever instantiations will allow us to go beyond this longstanding space-time barrier.

2 Preliminaries

2.1 Catalytic Model

Our machine model in this paper is an extension of ordinary space-bounded computation:

▶ **Definition 3** (Catalytic Turing machine). *A* catalytic Turing machine *is a Turing machine with four tapes: 1) a read-only* input *tape; 2) a write-only* output *tape; 3) a read-write* work *tape; and 4) a read-write* catalytic *tape.*

We say that a function $f : \{0,1\}^n \to \{0,1\}^m$ can be solved in $\mathsf{CSPACE}[s, c]$ *if there exists a catalytic Turing machine M whose input, output, work, and catalytic tapes have length n, m, s, and c respectively, such that for any $x \in \{0,1\}^n$ and $\tau \in \{0,1\}^c$, if M is initialized with x on the input tape and τ on the catalytic tape, then after execution M halts with $f(x)$ on the output tape and τ on the catalytic tape.*

We further say that f is in $\mathsf{CTIMESPACE}[t, s, c]$ *if M runs in time t for every x and τ.*

Since time is a consideration in our algorithm, we specify that in this paper we will use *Random Access Turing machines*, which allow us to jump with their heads to a particular tape cell, specified on an auxiliary *address* tape consisting of $\lceil \log(c + s) \rceil$ bits, as an atomic operation. This does not provide the machines with extra computational power, but only allows for faster simulation of RAM machines.

While our ultimate goal in catalytic computation is to compute a function into ordinary memory while leaving catalytic memory untouched, along the way it is necessary to alter the catalytic tape in predictable (and reversible) ways:

▶ **Definition 4** (Catalytic subroutine). *A* catalytic subroutine *is a program P whose net result is to 1) transform a section of the catalytic tape τ via addition, i.e. $\tau \mathrel{+}= v$ for some value v; 2) manipulate the free memory in any way; and 3) leave all other catalytic memory in its original configuration. We also require that every catalytic subroutine P has an inverse program P^{-1} whose result is to subtract v from the same region instead.*

Our algorithms will need to operate with catalytic memory containing values from a particular field $\mathbb{F}_p$. The details of this are left to the full version due to space constraints.

2.2 Function Statements

We first define all functions of interest in this paper:

▶ **Definition 5** (Connectivity). *For a directed graph $G = (V, E)$ and two vertices $s, t \in V$, we say s and t are* connected *if there is a path from s to t in G.*

▶ **Definition 6** (Longest common subsequence). *For a string $x \in \Sigma^*$, a* subsequence *is a string z that can be obtained by removing any number of characters of x.*

For two strings $x, y \in \Sigma^*$, the longest common subsequence *is the longest $z \in \Sigma^*$ that is a subsequence of both x and y, and we define the corresponding function* $\mathsf{LCS}(x, y) := |z|$.

▶ **Definition 7** (Edit distance). *Given a string $x \in \Sigma^*$, an* edit operation *is either a deletion, insertion or a substitution of a character in x.*

For two strings $x, y \in \Sigma^*$, their edit distance $\mathsf{ED}(x, y)$ *is the minimum number of edit operations needed to transform x into y.*

▶ **Definition 8** (Discrete Fréchet distance). *For two sequences $P = (p_1, \ldots, p_m), Q = (q_1, \ldots, q_n)$ of points from a metric space (M, d), their* monotone alignment *is a sequence of pairs $(i_k, j_k) \in [m] \times [n]$ of some length ℓ such that $(i_1, j_1) = (1, 1)$, $(i_\ell, j_\ell) = (m, n)$ and for all $2 \leq k \leq \ell$, $i_k \in \{i_{k-1}, i_{k-1} + 1\}$ and $j_k \in \{j_{k-1}, j_{k-1} + 1\}$.*

The discrete Fréchet distance *of P and Q is*

$$\mathsf{DFD}(P, Q) = \min_{A} \max_{(i,j) \in A} d(p_i, q_j),$$

where A ranges over all possible monotone alignments.

Informally, the discrete Fréchet distance describes the situation, where there are two frogs connected by a rope at points p_1, q_1, and in each step, each frog may jump forward to the next point in the sequence. The distance itself is then the minimum length of the rope such that the frogs can jump in a way that the rope does not snap.

In our algorithm, we assume that the metric d is represented by integers from the range $\{0, \ldots, \mathsf{poly}(n)\}$. This assumption is without loss of generality: there are $m \cdot n$ pairs in $P \times Q$, and it follows from the definition that the distance of one of the pairs is the discrete Fréchet distance, and our algorithm will only use the queries of the form (p_i, q_j) for some $p_i \in P, q_j \in Q$. Thus, we can represent the order of distances in the metric by simply ranking the distances between the pairs of points. These results are integers from the set $\{1, \ldots, m \cdot n\}$, which is sufficient. Provided the distances from the original metric can be compared using $\mathcal{O}(\log n)$ bits of space, we can translate from the metric into the ranks in logarithmic space and roughly quadratic time per query.

2.3 Pairwise Independent Hash Functions

Our algorithms for STCONN will require efficiently constructible *hash functions* for sparsifying our graph G:

▶ **Definition 9** (Pairwise Independent Hash Functions). *A family of functions $\mathcal{H}_{m,n} = \{h : [m] \to [n]\}$ is pairwise independent if for every $x_1 \neq x_2 \in [m]$, and every $y_1, y_2 \in [n]$, when a function h is chosen uniformly at random from $\mathcal{H}_{m,n}$,*

$$\Pr_{h \sim \mathcal{H}_{m,n}} [h(x_1) = y_1 \wedge h(x_2) = y_2] = \frac{1}{n^2}.$$

In addition, for any $h \in \mathcal{H}_{m,n}$, let $\mathsf{Im}(h)$ be the image of h.

It follows from Definition 9 that for every $x, y \in [m]$, $\Pr_{h \sim \mathcal{H}_{m,n}} [h(x) = y] = 1/n$.

Important for our result is that there exist pairwise independent hash functions that have a short description and can be constructed efficiently:

▶ **Lemma 10** ([69]). *For every $m, n \in \mathbb{N}$, there exists a pairwise independent hash function family $\mathcal{H}_{m,n}$, where a random function can be sampled from the family $\mathcal{H}_{m,n}$ using $r = \max\{\log(m), \log(n)\} + \log(n)$ bits.*

Moreover, given any $h \in \mathcal{H}_{m,n}$ (described by a string in $\{0,1\}^r$) and $x \in [m]$, $h(x)$ can be computed in time $\mathsf{poly}(\log(m, n))$ and space $\mathcal{O}(r)$.

2.4 Modular Arithmetic

In addition to hash functions, we will also require working over the integers modulo a prime p of logarithmic bit-size in an efficient manner:

▶ **Proposition 11.** *Given a prime p, we can compute addition, subtraction, and multiplication modulo p, in time* $\mathsf{poly}(\log(p))$ *and space* $\mathcal{O}(\log(p))$.

Working with modulo primes will be a useful proxy for testing whether or not a number is non-zero:

▶ **Proposition 12.** *Let $0 < m \leq 2^t$, and let $t \log t \leq p \leq 2t \log t$ be a randomly chosen prime. Then $m \not\equiv 0 \bmod p$ with probability 0.99 over p.*

Beyond testing for zero, we will sometimes need to represent numbers exactly, for which we use the *Chinese remainder representation* of a number x over many primes. There is a log-space procedure that can calculate bits of x in binary representation using *oracle* access to the Chinese remainder representation of x [54]:

▶ **Proposition 13** ([54])**.** *Let p_i denote the i-th prime. There is a procedure that works in space $\mathcal{O}(\log n)$ with query access to $x \bmod p_i$, for $i = 1, \ldots, n$, where $0 \leq x < 2^n$ is an arbitrary integer, that on input (n, j) represented in binary, where $n > j \geq 0$ are integers, outputs j-th bit of the binary representation of x. The procedure runs in polynomial time and makes in total $\mathcal{O}(n^2)$ queries to $x \bmod p_i$.*

We remark that the procedure would work as long as $x < \prod_{i=1}^{n} p_i$ which would provide $\mathcal{O}(\log n)$-factor savings in time and the number of queries. This is however irrelevant for us. Moreover the procedure can output all the bits of the binary representation of x in the usual order using the same space, time and number of queries. This could provide time savings of $\mathsf{poly}(n)$-factor for our edit distance and LCS algorithms.

3 Pushing Flow Through Walks

In this section we will utilize the short paths (or in our case, walks) decomposition from [20] as a generic setup for the flow-based algorithm of [43]. The latter assigned initial values to every node in the graph from the catalytic tape, and then "pushed" these values forward through the graph along every path via addition, using cancellation to collect up only terms from relevant (i.e., s–t) paths. To improve their catalytic space usage, we need to follow the color class framework of [20] and use recursion restricted to subsections of the graph.

▶ **Definition 14** (Walk matrix)**.** *Let $H = (V, E)$ be a graph on $m \in \mathbb{N}$ vertices, and for convenience assume m is a power of 2. Fix a field $\mathbb{F} = \mathbb{F}_p$ for some prime $p \leq \mathsf{poly}(m)$, and let $\mu : E \to \mathbb{F}$ be a weighting function on the edges of H. Given a walk W in H, we say the weight of W is the product of all edge weights in W with multiplicity, i.e. $\mu(W) = \prod_{e \in W} \mu(e)^{d_e}$ where d_e is the number of times e occurs in W.*

For any $\ell \in [\log m]$, and any $v_{in}, v_{out} \in V$, define $\Gamma_\ell(v_{in}, v_{out})$ to be the collection of all walks of length exactly 2^ℓ from v_{in} to v_{out} in H. We define the walk matrix *$\mathcal{W}_\ell \in \mathbb{F}^{V \times V}$ by*

$$\mathcal{W}_\ell[v_{in}, v_{out}] = \sum_{W \in \Gamma_\ell(v_{in}, v_{out})} \mu(W)$$

While we cannot store the walk matrix, we simplify our task in two ways: 1) we focus on a collection of input and output nodes, V_{in} and V_{out} respectively, of sufficiently small size to be stored; and 2) while we still cannot store the matrix $\mathcal{W}_\ell$ restricted to $V_{in} \times V_{out}$, we instead provide an input vector $\overrightarrow{\tau_{in}} \in \mathbb{F}^{V_{in}}$ and compute the output of $\overrightarrow{\tau_{in}} \cdot \mathcal{W}_\ell$ in the entries V_{out}.

▶ Remark 15. For convenience, given sets $V_{in}, V_{out} \subseteq V$ and a vector $\overrightarrow{\tau_{in}} \in \mathbb{F}^{V_{in}}$, we write $\overrightarrow{\tau_{in}} \cdot \mathcal{W}_\ell$ to mean the vector resulting from the product of matrix $\mathcal{W}_\ell$ with the (row) vector indexed by V which is $\overrightarrow{\tau_{in}}$ for all $v_{in} \in V_{in}$ and 0 everywhere else. If we add this to a vector $\overrightarrow{\tau_{out}} \in \mathbb{F}^{V_{out}}$, we mean that we restrict this vector to the entries indexed by $v_{out} \in V_{out}$ and do addition entry-wise by corresponding v_{out}.[3] Thus we have

$$(\overrightarrow{\tau_{in}} \cdot \mathcal{W}_\ell)[v_{out}] = \sum_{v_{in} \in V_{in}} \overrightarrow{\tau_{in}}[v_{in}] \cdot \sum_{W \in \Gamma_\ell(v_{in}, v_{out})} \mu(W)$$

Our V_{in} and V_{out} sets will be chosen via breaking V into easily computable blocks of equal size, often referred to as color classes.

▶ **Definition 16.** *Let H be a graph on $m \in \mathbb{N}$ vertices, and let $C \in [m]$ such that C and m are powers of 2. We partition V into sets $\{V_c\}_{c \in [C]}$, which we call* color classes*, such that $|V_c| = m/C$ for all c, by letting the first $\log C$ bits of any vertex v's label in $[m]$ indicate the color class of v.*

While the key property for our space and time bounds is the even distribution of nodes into classes, note that computing the color class and label within the color class of v can be done in time $\log C$ and $\log m/C$ respectively.

We now reach our main propagation subroutine which will drive all algorithms in this paper. Unlike [20] itself, we will require calculating each side of the recursion multiple times in order to get the cancellations necessary for [43]; however, by setting the parameters correctly we maintain an equivalent asymptotic runtime.

▶ **Lemma 17.** *Let $m \in \mathbb{N}$, let $\mathbb{F} = \mathbb{F}_p$ for some prime $p = \mathsf{poly}(m)$, and let $H = (V, E)$ be a graph on m vertices with edge weight function μ. Let $\mathcal{W}$ be defined with respect to H as per Definition 14 and let C and $\{V_i\}_{i \in [C]}$ be as per Definition 16.*

Assume that we have a catalytic subroutine P_0 which takes in $c_{in}, c_{out} \in [C]$ as well as sections $\overrightarrow{r_{in}}$, $\overrightarrow{r_{out}}$, $\overrightarrow{r_{mid}} \in \mathbb{F}^{m/C}$ from the catalytic tape, and adds $\overrightarrow{r_{in}} \cdot \mathcal{W}_0$ to $\overrightarrow{r_{out}}$.

Then there exists a catalytic subroutine P which takes in $\ell \in [\log m]$, $c_{in}, c_{out} \in [C]$, and sections $\overrightarrow{r_{in}}, \overrightarrow{r_{out}}, \overrightarrow{r_{mid}} \in \mathbb{F}^{m/C}$ from the catalytic tape, and whose effect is to add $\overrightarrow{r_{in}} \cdot \mathcal{W}_\ell$ to $\overrightarrow{r_{out}}$.

If P_0 runs in time T and uses free space S, then P runs in time $(4C)^\ell \cdot T$ and uses free space $\ell \log 4C + S$; it uses only the catalytic memory $\overrightarrow{r_{in}}, \overrightarrow{r_{out}}, \overrightarrow{r_{mid}}$ and the memory required to run P_0.

4 STCONN via Graph Decomposition

In this section we prove Theorem 1, i.e. deciding connectivity in an n-vertex graph. To do this, we start by discussing important parameterizations of Lemma 17 and what they say about the types of graphs and walks we can capture with them.

[3] Note that this definition is *not* the same as taking the product with submatrix $(\mathcal{W}_\ell)_{V_{in}, V_{out}}$; we will allow our walks to go via any vertices as long as they start in V_{in} and end in V_{out}.

In order to run in polynomial time we need to choose C and ℓ such that $(4C)^\ell \leq \mathsf{poly}(n)$, while for catalytic space $\frac{n}{2^{\Omega(\sqrt{\log n})}}$ we need to store $\overrightarrow{r_{in}}, \overrightarrow{r_{out}}, \overrightarrow{r_{mid}}$ using $(m/C) \cdot \mathcal{O}(\log n)$ bits, where m is the number of vertices of the graph in question. There are two extreme cases of interest:

- when $C = 1$, we can tolerate any 2^ℓ-length walk for $\ell \leq \log m$, which in particular can cover every vertex in the graph in question; with respect to this graph, however, we only get small catalytic space if $m \leq \frac{n}{2^{\Omega(\sqrt{\log n})}}$, i.e. working on graphs with many fewer vertices than n.
- when $\ell \leq \sqrt{\log n}$ and $C \leq 2^{\mathcal{O}(\sqrt{\log n})}$, we can only handle short walks, but in exchange we use sufficiently little catalytic storage even when working on large graphs, such as G or even larger graphs with $\mathcal{O}(n)$ vertices.

Thus we will have two phases of the algorithm as in [20]: 1) a *long walks* algorithm which works on a vertex-sparsified version of G whose edges represent walks of length λ in G; and 2) a *short walks* algorithm which runs on all of G but only for walks of length λ. For this we will use both of the extreme settings of Lemma 17 as discussed above. Due to space limitations, we postpone the proofs to the full version.

4.1 Long Walks via Graph Sparsification

In order to handle long walks in a graph, we need to reduce the number of vertices significantly while still not losing connectivity information about G. We do this through a randomized construction of a sufficiently large subset of vertices U, such that if any two vertices are connected in G, they are connected by taking short walks between the nodes in U. The following theorem may be of independent interest:

▶ **Theorem 18.** *There exists an algorithm $\mathcal{U}$ such that for every directed graph $G = (V, E)$ on n vertices, value $\lambda = \omega(\log(n))$, seed $\sigma \in \{0,1\}^{(6+\epsilon)\log(n)}$, and constant $\varepsilon > 0$, it satisfies the following properties:*

- *For every σ, there exists a multiset $U_\sigma \subset V$ of size $\mathcal{O}\left(\frac{n\log(n)}{\lambda}\right)$, such that on input $i \in [|U_\sigma|]$ in binary, $\mathcal{U}(1^n, \lambda, \sigma, \varepsilon, i)$ outputs the i^{th}-vertex of U_σ in time $\mathsf{poly}(\log(n))$ and space $\mathcal{O}(\log(n))$.*
- *With probability at least 0.99 over a uniformly random choice of σ, for every $u, v \in V$, if u is connected to v, then there exists an ℓ and a sequence of vertices $\tilde{\rho}_{u,v} = (u, s_1, s_2, \ldots, s_\ell, v)$, such that 1) $s_i \in U_\sigma$ for every $i \in [\ell]$, and 2) for each consecutive pair in $\tilde{\rho}_{u,v}$, there exists a path in G of length at most λ connecting them.*

In order to utilize any multiset $U \coloneqq U_\sigma$ given by Theorem 18, we will need to have access to connectivity at distance at most $2^{\Theta(\sqrt{\log n})}$ in G. Note that Lemma 17 counts walks of length *exactly* 2^ℓ for the given ℓ, but this can be easily rectified by adding self-loops, at the cost of obtaining a value which is not the exact count of short walks in the original graph.

▶ **Definition 19.** *Given graph $G = (V, E)$ on $n \in \mathbb{N}$ vertices, let G' be the graph obtained by adding self-loops edges to all vertices, i.e. $V' = V$ and $E' = E \cup \{(v, v)\}_{v \in V'}$.*

We construct $H = (V_U, E_U)$ to be the graph whose vertex set V_U is U with multiplicity plus the vertices s and t (for convenience we assume $|V_U|$ is a power of two, and otherwise we add dummy vertices to make it such) and whose edge set E_U is all $e = (v_{in}, v_{out}) \in V_U^2$ such that there exists a walk of length λ from v_{in} to v_{out} in G'. We define the weight function $\mu(e)$ for all $e \in E_U$ to be the number of walks of length λ from v_{in} to v_{out} in G'.[4]

[4] We alert the reader to the fact that for any $\mu(u, v)$ is *not* the number of u–v paths in G of length at most λ; it is at least as large as this value and can be much greater. This will not cause an issue to our analysis, since all we need is a non-zero lower bound for connected vertices as well as a cap on the total size of any weight.

We can now compute our long walks algorithm, which will be our global algorithm modulo a few cleanup steps:

▶ **Lemma 20.** *Assume we have access to graph $G = (V, E)$ on $n \in \mathbb{N}$ vertices, prime $p = \mathsf{poly}(n)$, and seed σ. Let $\mathbb{F} := \mathbb{F}_p$, define $\lambda := 2^{\sqrt{(\epsilon/2.1)\log n}}$, and let $U := U_\sigma$ be as given in Theorem 18. Define walk matrix $\mathcal{W}$ with respect to graph $H = (V_U, E_U)$ and weight function μ as defined in Definition 19.*

Assume we have access to a catalytic subroutine $\mathsf{SHORT}(V_{in}, V_{out}; \overrightarrow{r_{in}}, \overrightarrow{r_{out}})$, where we have $V_{in}, V_{out} \subseteq V_U$ and $\overrightarrow{r_{in}} \in \mathbb{F}^{V_{in}}, \overrightarrow{r_{out}} \in \mathbb{F}^{V_{out}}$ are any sections of the catalytic tape, whose effect is to add $\overrightarrow{r_{in}} \cdot \mathcal{W}_0$ to $\overrightarrow{r_{out}}$.

Then there exists a catalytic subroutine $\mathsf{LONG}(V_{in}, V_{out}; \overrightarrow{r_{in}}, \overrightarrow{r_{out}})$, where $V_{in}, V_{out} \subseteq V_U$ and $\overrightarrow{r_{in}} \in \mathbb{F}^{V_{in}}, \overrightarrow{r_{out}} \in \mathbb{F}^{V_{out}}$ are any sections of the catalytic tape, whose effect is to add $\overrightarrow{r_{in}} \cdot \mathcal{W}_\ell$ to $\overrightarrow{r_{out}}$, where $\ell := \log |V_U|$.

If SHORT uses time T, free space S, and catalytic space R, then LONG uses time $T \cdot \mathcal{O}(n^2)$, free space $S + \mathcal{O}(\log n)$, and catalytic memory $|V_{in}| + |V_{out}| + 3|V_U| \cdot \mathcal{O}(\log p)$, namely the vectors $\overrightarrow{r_{in}}, \overrightarrow{r_{out}}$ plus three additional catalytic vectors $\vec{r_1}, \vec{r_2}, \vec{r_3} \in \mathbb{F}^{V_U}$.

4.2 Short Walks via Color Classes

Our goal now is to compute SHORT. By the definition of the weights of H, we have that computing $\mathcal{W}_0$ with respect H is equivalent to computing $\mathcal{W}_{\log \lambda}$ with respect to G' plus accounting for multiplicities; thus we utilize Lemma 17 with this goal in mind.

▶ **Lemma 21.** *Assume we have access to graph $G = (V, E)$ on $n \in \mathbb{N}$ vertices, prime $p = \mathsf{poly}(n)$, and seed σ. Fix $\lambda := 2^{\sqrt{(\epsilon/2.1)\log n}}$, define graphs G' and H as per Definition 19, and define $\mathbb{F} := \mathbb{F}_p$ and $C := \lambda$.*

Then there exists a catalytic subroutine $\mathsf{SHORT}(V_{in}, V_{out}; \overrightarrow{r_{in}}, \overrightarrow{r_{out}})$, where $V_{in}, V_{out} \subseteq V_U$ are any sets of vertices of H and $\overrightarrow{r_{in}} \in \mathbb{F}^{V_{in}}, \overrightarrow{r_{out}} \in \mathbb{F}^{V_{out}}$ are any sections of the catalytic tape, whose effect is to add $\overrightarrow{r_{in}} \cdot \mathcal{W}_0$ to $\overrightarrow{r_{out}}$, where $\mathcal{W}$ is defined with respect to H.

SHORT runs in time $\mathcal{O}(|E| \cdot n^\epsilon)$, uses free space $\mathcal{O}(\log n)$, and uses catalytic memory $|V_{in}| + |V_{out}| + \frac{3n}{2^{\sqrt{(\epsilon/2.1)\log n}}} \cdot \mathcal{O}(\log p)$, namely the vectors $\overrightarrow{r_{in}}, \overrightarrow{r_{out}}$ plus three additional catalytic vectors $\vec{r_1}, \vec{r_2}, \vec{r_3} \in \mathbb{F}^{n/C}$.

4.3 Final Algorithm

Our LONG algorithm, using SHORT as a subroutine, gives us everything we need to compute STCONN, with the remaining issues being to choose U and p so as to ensure correctness and efficiency. We leave the proof details to the full version.

5 Edit Distance, Longest Common Subsequence and Discrete Fréchet Distance

In this section, we focus on computing edit distance, longest common subsequence and discrete Fréchet distance.

▶ **Theorem 22.** *The length of the longest common subsequence and edit distance of two strings of length n can be computed in time $\mathsf{poly}(n)$, using $\mathcal{O}(\log n)$ workspace and $\frac{n}{2^{\Omega(\sqrt{\log n})}}$ catalytic space.*

▶ **Theorem 23.** *Discrete Fréchet distance of two point sequences of length n with distance oracle whose values are in the range $\{0, \ldots, \mathsf{poly}(n)\}$ can be computed in time $\mathcal{O}(n^{4+\varepsilon})$, using $\mathcal{O}(\log n)$ workspace and $\frac{n}{2^{\Omega(\sqrt{\log n})}}$ catalytic space.*

All of these results follow from the same procedure for computing (weighted) reachability on a directed grid graph.

5.1 Grid graphs reachability

For the remainder of this section, we consider only fields of the form $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ for some prime p.

For the purpose of our algorithm, we extend the grid-like graphs used for computing the metrics to be layered, denoted by $\mathsf{Grid}_{n,n}$. Formally, for $n \in \mathbb{N}$ we define $\mathsf{Grid}_{n,n}$ as a directed graph such that

$$\begin{aligned} V(\mathsf{Grid}_{n,n}) &= \{(i,j) \colon 0 \le i \le n, 0 \le j \le n\}, \\ E(\mathsf{Grid}_{n,n}) &= \{((i,j),(i+1,j)) \colon 0 \le i \le n-1, 0 \le j \le n\} \cup \\ &\quad \{((i,j),(i+1,j+1)) \colon 0 \le i \le n-1, 0 \le j \le n-1\} \cup \\ &\quad \{((i,j),(i+1,j-1)) \colon 0 \le i \le n-1, 1 \le j \le n\}. \end{aligned}$$

Given a directed acyclic layered graph $G = (V, E)$ with weighted edges $\mu : V \times V \to \mathbb{N}$, we recall that the weight of an s,t-walk $W = (s = v_0, v_1, \ldots, v_k = t)$ is $\mu(W) = \prod_{i=0}^{k-1} \mu(v_i, v_{i+1})$. In fact, as we consider only layered graphs, and in particular $\mathsf{Grid}_{n,n}$, all walks from any u to any v are paths of the same length.

We thus define the total path weight[5] from s to t as $\mathcal{W}_{s,t} = \sum_{P\, s,t\text{-path}} \mu(P)$. In line with the flow-based algorithm of [43], our goal is to catalytically compute the total path weight from $(0,0)$ to (n,n) in a weighted Grid. To save space, we split each layer of the graph into a few color classes (which will be intervals in each of the layers), and then compute the total path weights recursively on the layers within the color classes up to the base case, where we will use only a single color class on which we will recurse slightly differently.

▶ **Theorem 24.** *There exists a catalytic algorithm that, given a prime $p = \mathsf{poly}(n)$, a graph $\mathsf{Grid}_{n,n}$ with weights $\mu : E(\mathsf{Grid}_{n,n}) \to \mathbb{N}$ given as an oracle $\mathcal{D}$ that returns the weights modulo p, and two vertices $u, v \in V(\mathsf{Grid}_{n,n})$ described by their indices in the layer such that the distance between u and v is a power of two, computes the value $\mathcal{W}_{u,v} = \sum_{P\, u,v\text{-path}} \mu(P) \bmod p$.*

The algorithm runs in time $\mathcal{O}(n^{3+\varepsilon})$ using free space $\mathcal{O}(\log n)$, catalytic space $\frac{n}{2^{\Omega(\sqrt{\log n})}}$ and $\mathcal{O}(\frac{n^{3+\varepsilon}}{2^{3\sqrt{\log n}}})$ weight oracle queries.

We remark that the assumption on the distance between u and v being a power of two is only made for simplicity. It is possible to support arbitrary distances at the cost of $\mathcal{O}(1)$ bits of free space per level of recursion, which would amount to $\mathcal{O}(\log n)$ additional free space.

Using Theorem 24 and well-known reductions to (weighted) reachability [16, 35, 49], Theorems 22 and 23 follow. Detailed proofs are available in the full version.

[5] One can also view this as $\overrightarrow{\tau_{in}}\mathcal{W}_\ell$ for a particular ℓ in Definition 14 with $\overrightarrow{\tau_{in}}$ set to be the unit vector corresponding to s.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *56th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2015)*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14.

2 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *48th Annual ACM Symposium on Theory of Computing (STOC 2016)*, pages 375–388, 2016. doi:10.1145/2897518.2897653.

3 Amir Abboud and Aviad Rubinstein. Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds. In *9th Innovations in Theoretical Computer Science Conference (ITCS 2018)*, volume 94 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:14, 2018. doi:10.4230/LIPIcs.ITCS.2018.35.

4 Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. Computing the discrete Fréchet distance in subquadratic time. *SIAM Journal on Computing*, 43(2):429–449, 2014. doi:10.1137/130920526.

5 Aryan Agarwala, Yaroslav Alekseev, and Antoine Vinciguerra. Linear matroid intersection is in catalytic logspace. In *17th Innovations in Theoretical Computer Science Conference (ITCS 2026)*, volume 362 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23, 2026. doi:10.4230/LIPIcs.ITCS.2026.3.

6 Aryan Agarwala and Ian Mertz. Bipartite matching is in catalytic logspace. In *66th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2025)*, 2025.

7 Miklós Ajtai, János Komlós, and Endre Szemerédi. Deterministic simulation in LOGSPACE. In *19th Annual ACM Symposium on Theory of Computing (STOC 1987)*, pages 132–140, 1987. doi:10.1145/28395.28410.

8 Yaroslav Alekseev, Yuval Filmus, Ian Mertz, Alexander Smal, and Antoine Vinciguerra. Catalytic computing and register programs beyond log-depth. In *50th International Symposium on Mathematical Foundations of Computer Science (MFCS 2025)*, volume 345 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:18, 2025. doi:10.4230/LIPIcs.MFCS.2025.6.

9 Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Grid graph reachability problems. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006)*, pages 299–313, 2006. doi:10.1109/CCC.2006.22.

10 Eric Allender and Klaus-Jörn Lange. RUSPACE($\log n$) $\subseteq$ DSPACE($\log^2 n/\log\log n$). *Theory of Computing Systems*, 31(5):539–550, 1998. doi:10.1007/S002240000102.

11 Carlos E. R. Alves, Edson N. Caceres, and Siang Wun Song. A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica*, 45(3):301–335, 2006. doi:10.1007/s00453-006-1216-z.

12 Alexandr Andoni, Michel Deza, Anupam Gupta, Piotr Indyk, and Sofya Raskhodnikova. Lower bounds for embedding edit distance into normed spaces. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 523–526, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644196.

13 Alexandr Andoni, Assaf Goldberger, Andrew McGregor, and Ely Porat. Homomorphic fingerprints under misalignments: Sketching edit and shift distances. In *45th Annual ACM Symposium on Theory of Computing (STOC 2013)*, pages 931–940, 2013. doi:10.1145/2488608.2488726.

14 Alexandr Andoni and Robert Krauthgamer. The computational hardness of estimating edit distance [Extended abstract]. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007)*, pages 667–676, 2007. doi:10.1109/FOCS.2007.60.

15 Alexandr Andoni and Robert Krauthgamer. The smoothed complexity of edit distance. In *35th International Colloquium on Automata, Languages, and Programming (ICALP 2008)*, volume 5125 of *Lecture Notes in Computer Science*, pages 357–369, 2008. doi:10.1007/978-3-540-70575-8_30.

16 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010)*, pages 377–386, 2010. doi:10.1109/FOCS.2010.43.

17 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. In *41st Annual ACM Symposium on Theory of Computing (STOC 2009)*, pages 199–204, 2009. doi:10.1145/1536414.1536444.

18 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *47th Annual ACM Symposium on Theory of Computing (STOC 2015)*, pages 51–58, 2015. doi:10.1145/2746539.2746612.

19 Nikhil Bansal, Moshe Lewenstein, Bin Ma, and Kaizhong Zhang. On the longest common rigid subsequence problem. *Algorithmica*, 56(2):270–280, 2010. doi:10.1007/s00453-008-9175-1.

20 Greg Barnes, Jonathan F. Buss, Walter L. Ruzzo, and Baruch Schieber. A sublinear space, polynomial time algorithm for directed *st*-connectivity. *SIAM Journal on Computing*, 27(5):1273–1282, 1998. doi:10.1137/S0097539793283151.

21 Tuğkan Batu, Funda Ergun, and Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 792–801, 2006. URL: https://dl.acm.org/doi/10.5555/1109557.1109644.

22 Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM Journal on Computing*, 21(1):54–58, 1992. doi:10.1137/0221006.

23 Sagar Bisoyi, Krishnamoorthy Dinesh, and Jayalal Sarma. On pure space vs catalytic space. *Theoretical Computer Science*, 921:112–126, 2022. doi:10.1016/J.TCS.2022.04.005.

24 Sagar Bisoyi, Krishnamoothy Dinesh, Bhabya Deep Rai, and Jayalal Sarma. Almost-catalytic computation. In *14th International Conference on Algorithms and Complexity (CIAC 2025)*, volume 15680 of *Lecture Notes in Computer Science*, pages 35–51, 2025. doi:10.1007/978-3-031-92935-9_3.

25 Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi HajiAghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, 2018. doi:10.1137/1.9781611975031.76.

26 Mahdi Boroujeni and Saeed Seddighin. Improved MPC algorithms for edit distance and ulam distance. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2019)*, pages 31–40, 2019. doi:10.1145/3323165.3323205.

27 Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly sub-cubic algorithms for language edit distance and RNA-folding via fast bounded-difference min-plus product. In *57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 375–384, 2016. doi:10.1109/FOCS.2016.48.

28 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2015)*, pages 79–97, 2015. doi:10.1109/FOCS.2015.15.

29 Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 1216–1235, 2018. doi:10.1137/1.9781611975031.79.

30 Kevin Buchin, Maike Buchin, and Joachim Gudmundsson. Detecting single file movement. In *16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2008)*, pages 1–10, 2008. doi:10.1145/1463434.1463476.

31 Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler, and Jun Luo. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications*, 21(3):253–282, 2011. doi:10.1142/S0218195911003652.

32 Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: Catalytic space. In *46th Annual ACM Symposium on Theory of Computing (STOC 2014)*, pages 857–866, 2014. doi:10.1145/2591796.2591874.

33 Harry Buhrman, Marten Folkertsma, Ian Mertz, Florian Speelman, Sergii Strelchuk, Sathyawageeswar Subramanian, and Quinten Tupker. Quantum catalytic space. In *20th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2025)*, volume 350 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:24, 2025. doi:10.4230/LIPIcs.TQC.2025.11.

34 Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. Catalytic space: Nondeterminism and hierarchy. *Theory of Computing Systems*, 62(1):116–135, 2018. doi:10.1007/S00224-017-9784-7.

35 Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *48th Annual ACM Symposium on Theory of Computing (STOC 2016)*, pages 712–725, 2016. doi:10.1145/2897518.2897577.

36 Diptarka Chakraborty and Raghunath Tewari. An $O(n^{\varepsilon})$ space and polynomial time algorithm for reachability in directed layered planar graphs. *ACM Transactions on Computation Theory*, 9(4):1–11, 2017. doi:10.1145/3154857.

37 Kuan Cheng, Alireza Farhadi, MohammadTaghi Hajiaghayi, Zhengzhong Jin, Xin Li, Aviad Rubinstein, Saeed Seddighin, and Yu Zheng. Streaming and small space approximation algorithms for edit distance and longest common subsequence. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 54:1–54:20, 2021. doi:10.4230/LIPIcs.ICALP.2021.54.

38 James Cook, Surendra Ghentiyala, Ian Mertz, Edward Pyne, and Nathan S. Sheffield. The structure of in-place space-bounded computation. *arXiv preprint*, arXiv:2510.12005, 2025. doi:10.48550/arXiv.2510.12005.

39 James Cook, Jiatu Li, Ian Mertz, and Edward Pyne. The structure of catalytic space: Capturing randomness and time via compression. In *57th Annual ACM Symposium on Theory of Computing (STOC 2025)*, pages 554–564, 2025. doi:10.1145/3717823.3718112.

40 James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *52nd Annual ACM Symposium on Theory of Computing (STOC 2020)*, pages 752–760, 2020. doi:10.1145/3357713.3384316.

41 James Cook and Ian Mertz. Trading time and space in catalytic branching programs. In *37th Computational Complexity Conference (CCC 2022)*, volume 234 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:21, 2022. doi:10.4230/LIPIcs.CCC.2022.8.

42 James Cook and Ian Mertz. Tree evaluation is in space $O(\log n \cdot \log\log n)$. In *56th Annual ACM Symposium on Theory of Computing (STOC 2024)*, pages 1268–1278, 2024. doi:10.1145/3618260.3649664.

43 James Cook and Edward Pyne. Efficient catalytic graph algorithms. In *17th Innovations in Theoretical Computer Science Conference (ITCS 2026)*, volume 362 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 43:1–43:22, 2026. doi:10.4230/LIPIcs.ITCS.2026.43.

44 Samir Datta, Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Randomized and symmetric catalytic computation. In *15th International Computer Science Symposium in Russia (CSR 2020)*, volume 12159 of *Lecture Notes in Computer Science*, pages 211–223, 2020. doi:10.1007/978-3-030-50026-9_15.

45 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 *56th Annual ACM Symposium on Theory of Computing (STOC 2024)*, pages 2039–2049, 2024. doi:10.1145/3618260.3649772.

46 Dean Doron, Edward Pyne, Roei Tell, and R. Ryan Williams. When connectivity is hard, random walks are easy with non-determinism. In *57th Annual ACM Symposium on Theory of Computing (STOC 2025)*, pages 1108–1117, 2025. doi:10.1145/3717823.3718303.

47 Roman Edenhofer. A space-space trade-off for directed *st*-connectivity. *arXiv preprint*, arXiv:2602.21088, 2026. doi:10.48550/arXiv.2602.21088.

48 Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. Tight lower bounds for *st*-connectivity on the NNJAG model. *SIAM Journal on Computing*, 28(6):2257–2284, 1999. doi:10.1137/S0097539795295948.

49 Thomas Eiter and Heikki Mannila. Computing discrete Fréchet distance. Technical Report CD-TR 94/64, Christian Doppler Laboratory for Expert Systems, TU Vienna, 1994.

50 Marten Folkertsma, Ian Mertz, Florian Speelman, and Quinten Tupker. Fully characterizing lossy catalytic computation. In *16th Innovations in Theoretical Computer Science Conference (ITCS 2025)*, volume 325 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:13, 2025. doi:10.4230/LIPIcs.ITCS.2025.50.

51 Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Unambiguous catalytic computation. In *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019)*, volume 150 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:13, 2019. doi:10.4230/LIPIcs.FSTTCS.2019.16.

52 Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Lossy catalytic computation. *arXiv preprint*, arXiv:2408.14670, 2024. doi:10.48550/arXiv.2408.14670.

53 Chetan Gupta, Raghunath Tewari, and Vimal Raj Sharma. Efficient isolation of perfect matching in $o(\log n)$ genus bipartite graphs. *arXiv preprint*, arXiv:2511.21217, 2025. doi:10.48550/arXiv.2511.21217.

54 William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002. doi:10.1016/S0022-0000(02)00025-9.

55 Sampath Kannan, Sanjeev Khanna, and Sudeepa Roy. STCON in directed unique-path graphs. In *28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 256–267, 2008. doi:10.4230/LIPIcs.FSTTCS.2008.1758.

56 Eamonn J. Keogh and Michael J. Pazzani. Scaling up dynamic time warping to massive datasets. In *3rd European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 1999)*, volume 1704 of *Lecture Notes in Computer Science*, pages 1–11, 1999. doi:10.1007/978-3-540-48247-5_1.

57 Man-Soon Kim, Sang-Wook Kim, and Miyoung Shin. Optimization of subsequence matching under time warping in time-series databases. In *2005 ACM Symposium on Applied Computing (SAC 2005)*, pages 581–586, 2005. doi:10.1145/1066677.1066814.

58 Masashi Kiyomi, Takashi Horiyama, and Yota Otachi. Longest common subsequence in sublinear space. *Information Processing Letters*, 168:106084, 2021. doi:10.1016/j.ipl.2020.106084.

59 Michal Koucký. Catalytic computation. *Bulletin of the EATCS*, 118, 2016. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/400.

60 Michal Koucký, Ian Mertz, Edward Pyne, and Sasha Sami. Collapsing catalytic classes. In *66th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2025)*, 2025.

61 Jiatu Li, Edward Pyne, and Roei Tell. Distinguishing, predicting, and certifying: On the long reach of partial notions of pseudorandomness. In *65th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2024)*, pages 1–13, 2024. doi:10.1109/FOCS61266.2024.00095.

62 Ian Mertz. Reusing space: Techniques and open problems. *Bulletin of the EATCS*, 141, 2023. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/780.

63 Edward Pyne. Derandomizing logspace with a small shared hard drive. *Computational Complexity*, 34:13, 2025. doi:10.1007/S00037-025-00275-6.

64 Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4):17:1–17:24, 2008. doi:10.1145/1391289.1391291.

65 Omer Reingold, Luca Trevisan, and Salil Vadhan. Pseudorandom walks on regular digraphs and the RL vs. L problem. In *38th Annual ACM Symposium on Theory of Computing (STOC 2006)*, pages 457–466, 2006. doi:10.1145/1132516.1132583.

66 Michael Saks and C. Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013)*, pages 1698–1709, 2013. doi:10.1137/1.9781611973105.122.

67 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.

68 Yakov Shalunov. Improved bounds on the space complexity of circuit evaluation. *arXiv preprint*, arXiv:2504.20950, 2025. doi:10.48550/arXiv.2504.20950.

69 Salil Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1–3):1–336, 2012. doi:10.1561/0400000010.

70 R. Ryan Williams. Simulating time with square-root space. In *57th Annual ACM Symposium on Theory of Computing (STOC 2025)*, pages 13–23, 2025. doi:10.1145/3717823.3718225.