A quick-and-dirty intro to CL

Nathan Sheffield

September 2025

1 Introduction

The complexity class CL is defined as the set of decision problems solvable by a machine with $O(\log(n))$ bits of "*clean*" workspace (that starts initialized with all 0s as usual), and an additional $\mathsf{poly}(n)$ bits of "*dirty*" workspace that starts *arbitrarily initialized*, and must be returned to the initial configuration at the end of computation. (Here, n is the length of the input, to which the algorithm has read-only access.) You can think of this as a process you've allocated a small amount of memory on your computer, but have allowed to arbitrarily mess up the contents on rest of your hard drive as long as it puts everything back when it's done. The original motivation for this model was to formalize the intuition that dirty space is useless — but the surprising discovery was that it *does* appear to be helpful sometimes!
Here, I'll tell you about CL algorithms for two problems we don't know how to solve without the dirty space. The first is a super elegant new algorithm by CL pros Ted and James for solving reachability (i.e. determining whether there's a path between a given pair of vertices) in directed graphs [CP25]. The second is a variant of the algorithm in the original CL paper, which evaluates "log depth threshold circuits" [BCKLS14].

2 CL algorithm for reachability in in directed graphs

Suppose you have a read-only representation of a directed graph on n vertices. Concretely, let's say the vertex set corresponds to the numbers $\{1, \dots, n\}$, and the graph representation just consists of an arbitrarily-ordered list of pairs (i, j), representing that there is an edge from i to j[1]. If you want to test whether there is a path from vertex 1 to vertex n, this is generally pretty easy to do: you can just run a breadth first (or depth first, whatever) search out of vertex 1 and see if you hit vertex n. However, this approach requires a large amount of available space: to do BFS, you need to keep track of whether each vertex has already been visited or not, so (on top of the read-only graph representation) the algorithm needs a workspace at least n bits large. A big open question is whether this problem can be solved with a workspace of size only $O(\log n)$. Here, we will see that we can solve it with only $O(\log n)$ bits of *clean* workspace, so long as we are also given a large amount of *dirty* workspace.

The approach is as follows: we divide up the dirty space into n "registers", one for each vertex. Let's say these registers are each r bits long, for some big r we'll decide on later. For every i, we'll think of the ith register as representing some number $x_i \bmod 2^r$. Note that each x_i starts as some arbitrary number, and we have to reset them to the same number at the end.

I'm now going to describe some operation we could perform on these registers if we so desired. Let's call it "*stirring the pot*". The procedure is: iterate through all of the edges of the graph (in the order they appear in the input), for each edge (i, j) replacing the value x_j with the new value $x_i + x_j \pmod{2^r}$. See Figure 1 for an example.

An important fact about this process is that it is reversible: if we were to instead iterate through the edges in the *opposite* order, and *subtract* the in-endpoint's register from the out-endpoint's register each

[1]If you're just worried about how much workspace you need and not worried about runtime, basically any reasonable ways of representing your graph turn out to be equivalent.

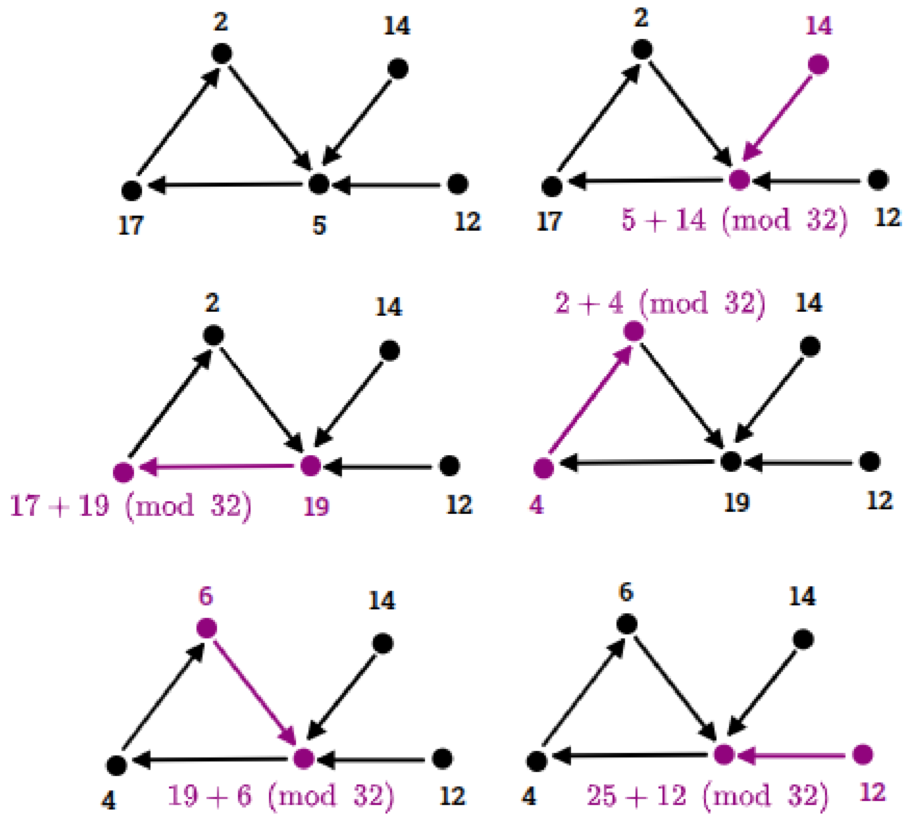

Figure 1. An example of "stirring the pot". Here, our registers are mod 32. Steps progress from left to right, then top to bottom. (The order in which edges are processed is arbitrary — again, we can just let this be determined by the order they appear in the input. All that matters is that it's some fixed order.)

time, we would exactly undo a single stirring of the pot. So if all we're doing to our dirty space is stirring, as long as we remember how many times we've stirred we will definitely be able to reset it at the end.

Some of you may now be asking: how on earth could this operation actually be a useful thing to do? That's indeed a fair question. The answer is: we're going to try stirring in two different ways and seeing if they give different results. First, suppose we just took the initial configuration and stirred the pot n times in a row. This will mess around with the register in some weird way, and eventually leave us with some number $x_i^{(\text{stirred})}$ in the ith register. Sure. But now, let's unstir everything and do it again — but this time, before our n rounds of stirring, we'll add 1 to x_1. Now, things will evolve in some different way, and we'll end up with different values $\widetilde{x}_i^{(\text{stirred})}$ for each vertex.

The crucial point is this: if there was no path from vertex 1 to vertex n, then $x_n^{(\text{stirred})}$ and $\widetilde{x}_n^{(\text{stirred})}$ should be exactly the same, since there's no route for x_1 to ever influence x_n. On the other hand, if there is a path, then after n rounds of stirring, eventually this 1 added to the 1st vertex will manage to propagate its way down to the nth vertex, resulting in some positive value being added to x_n on top of what would have happened in the original stirring. This bonus value need not be 1 of course — it will be something depending on the number of paths between them and the order in which we processed the edges. However, you can check that it will be at least 1, and at most (say) 2^{n^3}. So if we take $r > n^3$, this ensures that we will have $x_n^{(\text{stirred})} \neq \widetilde{x}_n^{(\text{stirred})}$ as long as there is no path from vertex 1 to vertex n.

So now the problem just becomes to determine whether $x_n^{(\text{stirred})} = \widetilde{x}_n^{(\text{stirred})}$ or not. This is not too hard to do. We can't fit either of these values in our workspace, so we can't store one and compare them directly. But what we can do is: stir n times, remember the *highest order bit* of $x_n^{(\text{stirred})}$, unstir n times, add 1 to x_1, stir n times, check if the highest order bit of $\widetilde{x}_n^{(\text{stirred})}$ equals that stored value, then unstir and subtract 1 from x_1 again. If we repeat this for the second highest order bit, third highest order bit, etc, we can check whether they agree on every bit (and hence whether there is a path from vertex 1 to vertex n) without ever having to store either of them in entirety in our clean space. Yay!

3 CL algorithm for evaluating log-depth threshold circuits

I think the previous story gives a good taste of what dirty space can do. But if you want some more, I can also tell you how to evaluate TC^1 circuits. Here, TC^1 refers to polynomial-size, logarithmic-depth, circuits of *exact value* gates. (An exact value gate outputs 1 if exactly t of its inputs are 1, and 0 otherwise, for some specified value t[2]). The set of problems solvable with such circuits is a pretty good first-order approximation of the set of problems we know how to solve in CL[3].

So then, let's get right into it! Say you've been handed (on read-only input) some circuit of depth $d = O(\log n)$ consisting of $S = \mathsf{poly}(n)$ many exact value gates, as well as values for the input wires of the circuit. You want to compute the value of the output wire. At your disposal is $O(\log n)$ bits of clean workspace and $\mathsf{poly}(n)$ bits of dirty workspace. What are you going to do?

Our procedure will consist of a long sequence of individual reversible "steps", each of which modifies the dirty space but doesn't read or store anything to the clean space. So the only clean space we'll need is the amount needed to perform a single individual one of these steps, plus enough for a counter to remember which step we're on. In particular, this means that as long as we can perform each step in logarithmic space, and we only take at most $\mathsf{poly}(n)$ steps (otherwise the counter is too big), the algorithm runs in CL[4].

The basic approach will be a type of recursion: we'll argue that if we can compute the outputs of all gates on level ℓ of the circuit in a small number of steps, then we can compute the outputs of all gates on level $\ell + 1$ with a larger number of steps. Of course, we can't actually *store* all of those outputs in the clear. So instead, our notion of "computing" the output of a gate will consist of *adding* its output value to a portion of the dirty space. I'll describe what I mean by this as follows.

Let's once again break up the dirty space into a whole bunch of registers, each of which we'll think of as holding a number mod S. For each gate g, we'll have some associated register $\mathtt{OUT}_g$, which is supposed to correspond to g's output. Now, we'll say that a sequence of steps "toggles g" if, regardless of the initial state of $\mathtt{OUT}_g$ prior to those steps, its value is incremented by 1 $\pmod S$ if gate g outputs 1 (and otherwise $\mathtt{OUT}_g$ remains unchanged). The meat of the algorithm comes from the following fact:

Lemma 3.1. Suppose there is a sequence P of reversible steps such that, after running P, we toggle all gates on layer ℓ of the circuit. Then, there is another sequence of at most $2|P| + 3S$ steps that toggles all gates on layer $\ell + 1$ of the circuit.

First, let's quickly note why this lemma is enough to let us win. Our circuit has depth d, and we can easily toggle the values of the input wires in a single step since we have them all written down in our read-

[2] A more standard definition of TC^1 would be to use majority gates, which check if at least half of the inputs are 1. This is equivalent up to polynomial size blowup and constant depth blowup. To simulate a majority gate using exact value gates, we use (#of input wires)/2 exact value gates checking if the number of 1s is exactly (#of input wires)/2, (#of input wires)/2 + 1, ..., (#of input wires). Then, we use another exact value gate to test if exactly one of those gates returns a 1.

[3] Although there has been a fair amount of recent progress finding other problems in CL — see e.g. [AM25; AAV25; AFMSV25].

[4] A reasonable concern you might raise is the following: even if this sequence of steps *exists*, it might not be the case that I can in small space *figure out* what I'm supposed to do on the ith timestep given just i. Fortunately, if you actually look at the procedure we describe it doesn't do anything super crazy, and so it's certainly "logspace uniform" in the sense that you can figure out the ith step to perform given the number i with logarithmic workspace.

only input. So, recursively applying Lemma 3.1 will give us a sequence of $O(S \cdot 2^d) = \mathsf{poly}(n)$ instructions that result in the output layer of the circuit being toggled. If we first record the least significant bit of $\mathtt{OUT}_g$ for the final output gate in our clean space, then perform this sequence of instructions, we can determine if it changed after toggling. Then, we can run the sequence of instructions in reverse, undoing them all and resetting the dirty space to its initial configuration. So, without further ado, let's go ahead and prove Lemma 3.1!

Proof of Lemma 3.1. In addition to $\mathtt{OUT}_g$, we'll also associate to each gate a bunch of extra helper registers $\mathtt{HELPER}_g^{(1)}, \ldots, \mathtt{HELPER}_g^{(S)}$. Our sequence of steps will be as follows:

i) For each gate g on layer $\ell + 1$ of the circuit, compute the sum (mod S) of the output registers corresponding to all of g's inputs. Letting i be this sum, replace register $\mathtt{OUT}_g$'s value with $\mathtt{OUT}_g - \mathtt{HELPER}_g^{(i)} \pmod S$.
ii) Run the sequence of steps P to toggle everything on layer ℓ.
iii) For each gate on layer $\ell + 1$, again compute the sum of the output registers corresponding to all of g's inputs. Subtract t from this sum (where g is an exactly t gate), take it mod S, and call that value j. Add 1 to $\mathtt{HELPER}_g^{(j)} \pmod S$.
iv) Undo P, un-toggling layer ℓ.
v) Once more, for each gate g add up the output registers of all its inputs and call this sum i. Replace $\mathtt{OUT}_g$'s value with $\mathtt{OUT}_g + \mathtt{HELPER}_g^{(i)} \pmod S$.

First, some bookkeeping. We ran P twice, and otherwise performed only 3 steps per gate on level $\ell + 1$. Each of these steps just consisted of computing a sum of a bunch of registers in the clean space, and then looking up the value of a register and adding something to it. This can easily be done with $O(\log n)$ clean workspace. Also, as each step is just an addition or subtraction, we can reverse the whole process to reset if desired. So assuming this sequence does in fact toggle level $\ell + 1$, we've won.

Now, what have we done here exactly? Well, observe that if exactly t of the input gates for a given gate g evaluate to 1, then once we toggle the registers on level ℓ, the sum of g's input registers will increase by t. So, the value of i on steps i) and v) will be exactly the same as the value of j on step iii). So, we will subtract off $\mathtt{HELPER}_g^{(i)}$, then increment $\mathtt{HELPER}_g^{(i)}$ by 1 before adding it back again — all in all successfully incrementing $\mathtt{OUT}_g$ by 1. On the other hand, if some other number of g's inputs evaluate to 1, then we will have $i \neq j$, so we will add and subtract the same value of $\mathtt{HELPER}_g^{(i)}$ and leave $\mathtt{OUT}_g$ unchanged. ■

4 Final notes

These are cute algorithms! But I think on some deep level we still don't exactly understand *why* they work. Nobody really knows what all we should expect to be able to do with dirty space or how powerful we should expect CL to be. That's part of why I think this is a super exciting area to work on! Go forth and prove new theorems. But before you go, let me just tell you a couple of the most important things we know about CL:

i) The power of CL remains unchanged if you allow the algorithm to use randomness [CLMP25], use nondeterminism [KMPS25], or incorrectly reset a constant number of the dirty bits [GJST24].
ii) We know anything in CL can be solved by a randomized algorithm in *expected* polynomial time [BCKLS14]. But we don't know how to show that $\mathsf{CL} \subseteq \mathsf{P}$. The best containment we know is in a somewhat exotic superclass of P called "LOSSY", which captures the power of compression arguments [CLMP25].
iii) In fact, there is some oracle relative to which $\mathsf{CL}^O = \mathsf{EXPTIME}^O$ [CGMPS25].
iv) Thinking about CL can help even when you don't have dirty space — see e.g. the Cook–Mertz tree evaluation algorithm [CM24], which enabled simulation of $\mathsf{TIME}[T]$ multi-tape TMs in $\mathsf{SPACE}[\sqrt{T \log T}]$ [Wil25].

For a more in-depth survey, I very highly recommend the one by Ian Mertz, which helped get me into this area 2 years ago [Mer+23]!

References

[AAV25] Aryan Agarwala, Yaroslav Alekseev, and Antoine Vinciguerra. “Linear Matroid Intersection is in Catalytic Logspace”. In: *arXiv preprint arXiv:2509.06435* (2025).

[AFMSV25] Yaroslav Alekseev, Yuval Filmus, Ian Mertz, Alexander Smal, and Antoine Vinciguerra. “Catalytic computing and register programs beyond log-depth”. In: *arXiv preprint arXiv:2504.17412* (2025).

[AM25] Aryan Agarwala and Ian Mertz. “Bipartite matching is in catalytic logspace”. In: *arXiv preprint arXiv:2504.09991* (2025).

[BCKLS14] 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.* 2014, pp. 857–866.

[CGMPS25] James Cook, Surendra Ghentiyala, Ian Mertz, Edward Pyne, and Nathan S. Sheffield. “Computing with a Full Input: Structural Complexity of In-Place Computation”. Unpublished manuscript; under preparation. 2025?

[CLMP25] James Cook, Jiatu Li, Ian Mertz, and Edward Pyne. “The structure of catalytic space: Capturing randomness and time via compression”. In: *Proceedings of the 57th Annual ACM Symposium on Theory of Computing.* 2025.

[CM24] James Cook and Ian Mertz. “Tree Evaluation Is in Space O(log n log log n)”. In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing.* 2024, pp. 1268–1278.

[CP25] James Cook and Edward Pyne. “Efficient Catalytic Graph Algorithms”. In: *arXiv preprint arXiv:2509.06209* (2025).

[GJST24] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. “Lossy catalytic computation”. In: *arXiv preprint arXiv:2408.14670* (2024).

[KMPS25] Michal Koucký, Ian Mertz, Edward Pyne, and Sasha Sami. “Collapsing catalytic classes”. In: *2025 IEEE 66th Annual Symposium on Foundations of Computer Science (FOCS).* 2025.

[Mer+23] Ian Mertz et al. “Reusing space: Techniques and open problems”. In: *Bulletin of EATCS* 141.3 (2023).

[Wil25] R Ryan Williams. “Simulating time with square-root space”. In: *Proceedings of the 57th Annual ACM Symposium on Theory of Computing.* 2025, pp. 13–23.