CS 860: Space Bounded Computation **University of Waterloo: Winter 2026**

Lecture 4: Savitch's Theorem and Directed Reachability in Small Space

February 3, 2026

Instructor: Sepehr Assadi

Topics of this Lecture

In the last lecture, we studied the *undirected* s-t connectivity problem. In this lecture, we switch to the *directed* version of the same problem.

1 The STCONN problem

Our problem of interest in this lecture is the following.

Problem 1. In the s-t connectivity (or reachability) problem, STCONN for short, we have an n-vertex directed graph $G = (V, E)$ and two vertices s and t as the input and the goal is to output whether or not s can reach t in G.

Recall that this problem can be easily solved in $O(n \log n)$ space and $O(n + m)$ time on n-vertex m-edge graphs using any BFS/DFS approach (it is also easy to solve this problem in $O(n)$ space and $O(mn)$ time). Our goal in this lecture is to examine much more space-efficient algorithms for this problem.

We start by presenting Savitch's theorem [Sav70] that gives an $O(\log^2 n)$ space algorithm for STCONN. This algorithm runs in $n^{O(\log n)}$ time, namely, has *quasi-polynomial* running time. We then study the longstanding question of obtaining a sublinear-space algorithm for STCONN that runs in *polynomial* time, and present the state-of-the-art result by [BBRS92]—from already more than 30 years ago—that achieve a polynomial time algorithm in $n/2^{O(\sqrt{\log n})}$ space.

2 Savitch's Theorem for STCONN

We start with the celebrated Savitch's theorem.

Theorem 1 ([Sav70]). STCONN *can be solved in* $O(\log^2 n)$ *space.*

The proof of Theorem 1 at this point is a staple in most complexity theory textbooks and courses. Consider the specification for an algorithm, which we call `Savitch` (we assume the input graph is $G = (V, E)$ and is fixed at this point and we do not specify it as input to the algorithm explicitly):

- `Savitch`(u, v, D): the input is vertices $u, v \in V$ and integer $D \geqslant 0$; the output is whether or not u can reach v in at most D hops (edges).

The answer for STCONN on an n-vertex graph G and vertices s and t, is to return `Savitch`$(s, t, n-1)$ since if s can reach t, it can do so with $n-1$ edges at most. We now give an implementation of `Savitch`.

Algorithm 1. `Savitch`(u, v, k):

```
1. If $D = 0$, return YES if $u = v$ and NO otherwise.
2. If $D = 1$, return YES if $u = v$ or $(u, v)$ is an edge, and NO otherwise.
3. For any vertex $w \in V$:
       if Savitch$(u, w, \lfloor D/2 \rfloor)$ and Savitch$(w, v, \lceil D/2 \rceil)$ both return YES, return YES.
4. If an answer is not returned at this point, return NO.
```

We now analyze this algorithm.

Correctness. The base cases of $D = 0$ and $D = 1$ are handled directly and are easy to verify are correct immediately. For the induction step, if u can reach v in D hops, then there is a vertex w "in the middle" where u can reach w in $\lfloor D/2 \rfloor$ hops and w can reach v in another $\lceil D/2 \rceil$ hops. Since the algorithm enumerates all choices for w, it will return YES correctly. Conversely, if the algorithm returns YES, it must have found a vertex w that u can reach in $\lfloor D/2 \rfloor$ hops and w can reach v in $\lceil D/2 \rceil$ hops and so u can reach v in at most $\lfloor D/2 \rfloor + \lceil D/2 \rceil = D$ hops, so YES was the correct answer.

Space complexity. The depth of the recursion in `Savitch`$(\cdot, \cdot, D)$ is $O(\log D)$ as each recursive call reduces D by a factor of 2. Moreover, for each recursive call, we need to store which step of `Savitch` we are in to continue after the recursion, which requires us to specify the vertex w (plus $O(1)$ extra bits) and thus we need to store $O(\log n)$ bits per level of the recursion. Thus, the total space of `Savitch`$(\cdot, \cdot, D)$ is $O(\log D \cdot \log n)$. Since we are running the algorithm at the end with $D = n - 1$, we get a total of $O(\log^2 n)$ space.

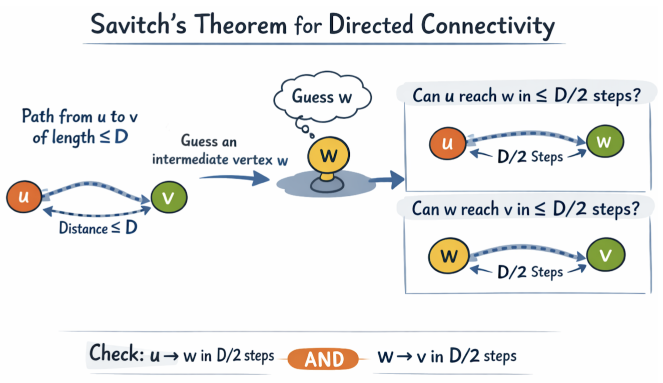
Savitch's Theorem for Directed Connectivity
Path from u to v of length ≤ D
u
v
Distance ≤ D
Guess an intermediate vertex w
Guess w
W
Can u reach w in ≤ D/2 steps?
u
w
D/2 Steps
Can w reach v in ≤ D/2 steps?
W
V
D/2 Steps
Check: u → w in D/2 steps
AND
w → v in D/2 steps

Figure 1: An illustration of the `Savitch` algorithm (thanks to ChatGPT).

It is worth noting that in each step of the recursion, we are spending $O(n)$ time also, so the total runtime is $n^{O(\log D)} = n^{O(\log n)}$ time. This concludes the proof of Theorem 1.

Remark. Savitch's theorem is generally stated (and was proved) as showing that a non-deterministic $f(n)$-space algorithm can be simulated by a deterministic $O(f(n)^2)$-space algorithm, i.e.,

$$\mathsf{NSpace}\left(f(n)\right) \subseteq \mathsf{Space}\left(f(n)^2\right).$$

We will get back to this in future lectures once we defined non-deterministic space computation.

3 STCONN in Polynomial-Time and Sublinear-Space?

A longstanding open question, in light of Theorem 1, is whether we can solve STCONN in space much smaller than $O(n)$ (needed by DFS/BFS) and yet polynomial time.

At this point, for all we know, it is possible that STCONN is in L, i.e., can be solved in $O(\log n)$ space and polynomial time[1]. Such a result however is considered unlikely as it implies that $\mathsf{NL} = \mathsf{L}$, i.e., non-deterministic logspace is the same as logspace – given we have not even defined non-determinism yet, we do not go into more details beside saying that most complexity theorists believe $\mathsf{NL} \neq \mathsf{L}$. By the same token, proving that STCONN cannot be solved in $O(\log n)$ space will imply $\mathsf{NL} \neq \mathsf{L}$ yet another major breakthrough in complexity theory[2], which appears to be out of reach.

The above discussion does not preclude us from hoping for any algorithm with, say, $n^{0.99}$ space and polynomial time. Yet, even no such algorithm is known. The best result on this front—already more than 30 years old—is the following theorem of Barnes, Buss, Ruzzo, and Schieber [BBRS92].

Theorem 2 ([BBRS92])**.** STCONN *can be solved in* $n/2^{\Theta(\sqrt{\log n})}$ *space and polynomial time.*

We present a randomized variant of Theorem 2 following the same overall strategy. The algorithm consists of two parts: (1) a less space-efficient but more time-efficient version of Savitch's theorem for handling "short paths" and (2) a *shortcutting* algorithm for handling "long paths".

3.1 Part One: Savitch's Algorithm, Faster but With More Space

We design a variant of `Savitch`, which we call `Savitch++`, that uses more space but can be more time-efficient than the original `Savitch` for solving STCONN. Specifically, we will prove the following lemma.

Lemma 3. *There exists an algorithm that given any n-vertex graph $G = (V, E)$, vertices $s, t \in V$, and integers $K, D \geqslant 1$ (potentially a function of n), determines whether or not s can reach t with at most D hops using $O((n/K + \log n) \cdot \log D)$ space and $K^{O(\log D)} \cdot \mathrm{poly}(n)$ time.*[3]

Let $V_1 \sqcup \ldots \sqcup V_k$ be any arbitrary partitioning of the vertices V of G into K equal-size groups such that $s \in V_1$ and $t \in V_K$. Consider the following specification for `Savitch++`:

- `Savitch++`(x, i, j, D): the input is a string $x \in \{0,1\}^{n/K}$, and integers $i, j \in [K]$, $D \geqslant 0$; the output is a string $y \in \{0,1\}^{n/K}$ with the following meaning:

  Let S be the subset of V_i whose characteristic vector is x (i.e., the ℓ-th vertex of V_i, in the lexicographic order, is in S iff $x_i = 1$). Let T be the subset of V_j that are all vertices which can be reached from *some* vertex in S with at most D hops. Then, y is the characteristic vector of T.

[1] Recall that *all* $O(\log n)$-space algorithms are polynomial time (as long as they terminate).

[2] Informally, and not at all accurately in any shape or form, think of proving $\mathsf{NL} \neq \mathsf{L}$ as the "space variant and (distant) cousin" of proving $\mathsf{NP} \neq \mathsf{P}$.

[3] We note that in this result, the main term in the space is $O((n \log D)/K)$ and the main term in the runtime is $K^{O(\log D)}$.

With this specification in mind, we can test if s can reach t in at most D hops as follows. Let x be the vector that is zero everywhere except for index corresponding to vertex $s \in V_1$; run $\texttt{Savitch++}(x, 1, K, D)$ and let y be the resulting vector; then, check if the index corresponding to vertex $t \in V_K$ in y is one or not.

We now implement the algorithm.

```
Algorithm 2. Savitch++($x, i, j, D$):

  1. Initialize $y \leftarrow 0^{n/K}$.
  2. If $D = 0$, return $x$ if $i = j$ and $y$ otherwise,
  3. If $D = 1$, for $u \in V_i$ and $v \in V_j$:
         if $u = v$ (which can only happen when $i = j$) or $(u, v)$ is an edge, set $y_v = 1$.
     Return $y$.
  4. If $D > 1$, for $\ell = 1$ to $K$:
         Let $x' = \texttt{Savitch++}(x, i, \ell, \lfloor D/2 \rfloor)$ and $x'' = \texttt{Savitch++}(x, \ell, j, \lceil D/2 \rceil)$. Update $y \leftarrow y \vee x''$.
     Return $y$.
```

We now analyze this algorithm.

Correctness. The proof of correctness is almost identical to that of $\texttt{Savitch}$. The base cases follow by a direct inspection. For the induction step, suppose a vertex v in V_j is reachable from some vertex u in V_i whose corresponding bit $x_u = 1$. In this case, we need $y_v = 1$ also. But this happens because there is a vertex w in V, and thus some V_ℓ, such that u (and thus x "in" V_i) can reach w in at most $\lfloor D/2 \rfloor$ hops and the vertex w (and thus x'' in V_ℓ) can reach v in another $\lceil D/2 \rceil$ step. Thus, the algorithm will set $y_v = 1$ in this case (and since it is taking OR of all ℓ's, the answer remains correct). The converse part, namely, that $y_v = 1$ only if v is reachable from x in V_i also holds exactly for the same reason.

Space complexity. The depth of recursion in the algorithm is $O(\log D)$ as before. At each level of recursion, the algorithm needs to maintains intermediate values of ℓ and y, x', x'' plus $O(1)$ for the line of code, and thus needs $O(n/K + \log n)$ bits. Hence, the space complexity is $O((n/K + \log n) \cdot \log D)$ as desired.

Time complexity. Let $T(D)$ denote the worst-case runtime of $\texttt{Savitch++}(\cdot, \cdot, \cdot, D)$. Then, for $D > 1$,

$$T(0) = O(1) \qquad \text{(base case of returning } x \text{ directly)}$$
$$T(1) = O((n/K)^2) \qquad \text{(base case of enumerating over } V_i \times V_j \text{ of size } (n/K)^2\text{)}$$
$$T(D) \leqslant K \cdot (2 \cdot T(D/2) + O(n/K)).$$
$$\text{(enumerating over } K \text{ choices for } \ell\text{, each involving two recursive call on } D/2 \text{ and updating } y\text{)}$$

This recursion tree has depth $O(\log D)$, a branching factor of $2K$ at each node, and poly(n) work per node. So, the total number of nodes is $K^{O(\log D)}$ and the total work is $K^{O(\log D)} \cdot \text{poly}(n)$ as desired.

This concludes the proof of Lemma 3.

For our purpose, by setting

$$K = 2^{\sqrt{\log n}} \qquad \text{and} \qquad D = 2^{\sqrt{\log n}} \tag{1}$$

in Lemma 3, we get an algorithm with space and time complexity, respectively

$$\frac{n}{2^{\Theta(\sqrt{\log n})}} \qquad \text{and} \qquad \left(2^{\sqrt{\log n}}\right)^{O(\sqrt{\log n})} \cdot \text{poly}(n) = \text{poly}(n). \tag{2}$$

That is, we can solve STCONN in polynomial time and $n/2^{\Theta(\sqrt{\log n})}$ space as long as the distance between s and t is at most $D = 2^{\sqrt{\log n}}$. Thus, this algorithm can handle the "short path" part, alluded to earlier.

3.2 Part Two: Shortcutting for Long Paths

We now design the second part of the main algorithm in Theorem 2. This algorithm relies, in a blackbox way, on any subroutine that can solve STCONN for "short paths" to solve the general problem. Plugging in our algorithm from Lemma 3 with parameters in Eq (1) for this subroutine then allows us to conclude the proof of Theorem 2.

Lemma 4. *Let $\mathbb{A}(D)$ be any algorithm for* STCONN *as long as the distance between s and t is at most D.*

There exists a randomized algorithm that given any n-vertex graph $G = (V, E)$, vertices $s, t \in V$, and integer $D \geqslant 1$ (potentially a function of n), with high probability[4] *behaves as follows. It determines whether or not s can reach t (regardless of the length of the path) using* $\text{poly}(n)$ *calls to $\mathbb{A}(D)$ and additional* $\text{poly}(n)$ *time, and $O(n \log^2 n/D)$ space plus the space complexity of $\mathbb{A}(D)$.*

The idea behind the algorithm is pretty simple. We sample $\approx n/D$ random vertices S which means that for a fixed s-t path (assuming it exists), with high probability, we have sampled one vertex every $\leqslant D$ steps. We then start from s and find all vertices reachable in S from s in D steps using $\mathbb{A}(D)$. This allows us to "shortcut" the first D edges of the fixed s-t path. We then continue from these vertices and do another round of $\mathbb{A}(D)$ to shortcut the next D edges. Continuing this for $O(n/D)$ steps thus takes us from s to t. We now formalize the algorithm.

Algorithm 3.

1. Let $S \subseteq V$ obtained by sampling each vertex in $V \setminus \{s, t\}$ independently and with probability $p := 4 \log n/D$. Add s and t to S additionally.
2. Let $U = \{s\}$ initially.
3. For $i = 1$ to $n/D + 1$ and $v \in S$ (in this order):
   (a) Run $\mathbb{A}(D)$ on the graph obtained from G by connecting a new vertex s' to all vertices in U; the goal is to check if there is a path from the starting vertex s' to the target vertex v.
   (b) If the algorithm returns YES (i.e., v is reachable from s' in at most D hops), add v to U.
4. Return YES if t belongs to U and NO otherwise.

Correctness. Firstly, all vertices in the set U throughout the algorithm are reachable from the vertex s: a vertex is only added to U if it is reached by some other vertex in U and initially U only contains s. Thus, if s cannot reach t in G, the algorithm always returns NO correctly. Now, suppose s can reach t in G.

Consider a fixed path P from s to t in G. Partition this path into n/D subpaths $P_1, P_2, \ldots, P_{n/D}$ containing the first $D/2$, then second $D/2$, and so on vertices in P. For any $i \in [n/D]$, we have

$$\Pr\left(\text{no vertex from } P_i \text{ are sampled in } S\right) = (1-p)^{D/2} \leqslant \exp\left(-\frac{4\log n}{D} \cdot D/2\right) \leqslant n^{-2}.$$

Thus, by union bound over n/D choices, with high probability, there is one vertex per each P_i in S. Let these vertices be $v_1, v_2, \ldots, v_{n/D}$ and define $v_0 = s$ and $v_{n/D+1} = t$. Note that for all $i \in [n/D + 1]$, there is a path of length at most D from v_{i-1} to v_i in G.

[4]Here, and throughout this course (and almost any time you are reading a paper in theoretical computer science), the term 'with high probability' means with probability $1 - 1/\text{poly}(\text{input size})$ for some polynomial (almost always, as is the case in our lemma, the degree of the polynomial can be made arbitrarily large as well).

We now claim, by induction, that after iteration i of the for-loop in Algorithm 3, vertex v_i belongs to the set U. For v_0, this holds immediately by definition. Suppose this is true for some i. During the iteration $i+1$, we know v_i belongs to U and since $\mathbb{A}(D)$, by iterating over all choices of $v \in S$, finds all vertices in S that are reachable from U with at most D hops, we will add v_{i+1} to U in this iteration, proving the induction hypothesis.

Thus, at the end, t belongs to the set U and the algorithm returns the correct answer in this case also.

Space complexity. The space complexity of the algorithm consists of the following:

- the space needed to store the set S, which with high probability, has $O(n \log n/D)$ vertices and thus $O(n \log^2 n/D)$ bits suffices to store it;
- the space needed to store U which is at most the same as S;
- $O(\log n)$ space for iterating over $i \in [n/D]$ and $v \in S$ in the for-loops;
- the space needed by $\mathbb{A}(D)$.

Since $D \leqslant n$ always, this is $O(n \log^2 n/D)$ space plus the space of $\mathbb{A}(D)$ as desired.

Time complexity. The algorithm involves running $\mathbb{A}(D)$ for $O(n/D) \cdot O(n \log n/D)$ times in a for-loop, plus poly(n) extra time for other computations. So, the runtime is poly(n) times $\mathbb{A}(D)$'s time complexity.

Remark. [BBRS92] presents a deterministic algorithm with guarantees the same as the ones in Lemma 4, again using $\mathbb{A}(\cdot)$ as a subroutine. That algorithm is also not particularly difficulty in any way but we opted for the current randomized algorithm in this lecture as its ideas are more transparent.

3.3 Concluding the Proof of Theorem 2, Albeit with Randomization

Consider the subroutine $\mathbb{A}(\cdot)$ in Lemma 4. We will set parameters K, D as in Eq (1) to get an implementation of $\mathbb{A}(D)$ in $n/2^{\Theta(\sqrt{\log n}}$ and poly(n)-time. Moreover, since $D = 2^{\sqrt{\log n}}$, Lemma 4 gives us an algorithm with poly(n) running time and space of

$$O(\frac{n \log^2 n}{D}) + \frac{n}{2^{\Theta(\sqrt{\log n})}} = \frac{n}{2^{\Theta(\sqrt{\log n})}}.$$

This gives a randomized algorithm for STCONN that achieves the guarantees of Theorem 2 with high probability. We refer the reader to [BBRS92] for the full proof of the deterministic algorithm.

References

[BBRS92] Greg Barnes, Jonathan F. Buss, Walter L. Ruzzo, and Baruch Schieber. A sublinear space, polynomial time algorithm for directed s-t connectivity. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference, Boston, Massachusetts, USA, June 22-25, 1992*, pages 27–33. IEEE Computer Society, 1992. 1, 3, 6

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970. 1, 2