CS 466/666 Algorithm Design and Analysis . Spring 2026

Lecture 7 : Graph Sketching

Graph sketching is a technique to compress a graph while supporting queries and updates.

This is very useful in designing streaming algorithms (as well as traditional algorithms) for graph problems.

Tools for Graph Sketching

The goal in this lecture is to introduce graph sketching techniques to design semi-streaming algorithms (using $\tilde{O}(n)$ space) for graph problems such as computing a spanning forest, k-edge-connectivity, etc.

To do so, we use tools from hashing to build the following primitives.

- Sparse recovery: Construct a sketch $sk(x) \in \mathbb{Z}^{\tilde{O}(s)}$ such that if $x \in \mathbb{Z}^n$ is s-sparse then we can recover all entries of x by reading $sk(x)$ in $\tilde{O}(s)$ time; if x is not s-sparse then return "not s-sparse".
- l_0-Sampling: Construct a sketch $sk(x) \in \mathbb{Z}^{\text{polylog}(n)}$ such that we can sample a non-zero of x from $sk(x)$ in polylog(n) time with high probability.

Linear Sketch: The sketches that we will construct satisfy the properties that $sk(c \cdot x) = c \cdot sk(x)$ for any scalar c, and $sk(x+y) = sk(x) + sk(y)$ for any x, y.

In other words, $sk(x)$ is a linear map.

So, if $sk(x)$ is a k-dimensional vector, then there is a $k \times n$ matrix M such that $sk(x) = Mx$.

But, we won't store M as this requires too much space.

Instead, M will be defined implicitly using hash functions that require very little space.

One important advantage of having linear sketches are for distributed computing, where each computer has part of the input, and they can compute the sketch locally and add them up to obtain the sketch as $sk(x) = sk(x^{(1)} + \ldots + x^{(l)}) = sk(x^{(1)}) + \ldots + sk(x^{(l)})$.

1-Sparse Recovery

First we consider the simple setting of recovering a 1-sparse x.

We will then mention how this can be extended to recover an s-sparse x.

And then we will use it to build a linear sketch for l_0-sampling, which will be used for graph sketching.

Algorithm 1

Maintain 3 numbers $w_1 = \sum_{i=1}^{n} x_i$, $w_2 = \sum_{i=1}^{n} i \cdot x_i$, and $w_3 = \sum_{i=1}^{n} i^2 \cdot x_i$.

Clearly, these can be updated easily. Given (i, Δ), update $w_1 \leftarrow w_1 + \Delta$, $w_2 \leftarrow w_2 + i \cdot \Delta$, and $w_3 \leftarrow w_3 + i^2 \cdot \Delta$.

The idea is that w_1 and w_2 are used to recover x if x is 1-sparse, and w_3 is used to check whether x is 1-sparse or not.

Clearly, if x is 1-sparse where $x_j \neq 0$, then $w_1 = x_j$ and $w_2 = j \cdot x_j$, and so $(j, x_j) = (w_2/w_1, w_1)$.

Also, if x is 1-sparse, then we expect $w_3 = j^2 \cdot x_j = \left(\frac{w_2}{w_1}\right)^2 w_1 = w_2^2/w_1$.

It turns out that $w_3 = \left(\frac{w_2}{w_1}\right)^2 w_1$ if and only if x is 1-sparse for $x \geq 0$.

A nice way to see this is that w_3/w_1 is the second moment of the random variable X, where $X = i$ with probability $x_i / \sum_j x_j$, while w_2/w_1 is the first moment of X.

Then, we know that $w_3/w_1 = \mathbb{E}[X^2] \geq \left(\mathbb{E}[X]\right)^2 = \left(\frac{w_2}{w_1}\right)^2$, and the inequality is tight only when x is deterministic, i.e. $X = \mathbb{E}[X]$ always and so x is 1-sparse.

So, to answer queries, if $w_3 \neq \left(\frac{w_2}{w_1}\right)^2 w_1$ then return "not 1-sparse"; otherwise, return $(j, x_j) = (w_2/w_1, w_1)$.

The advantage of this algorithm is that this is deterministic (i.e. always correct), but the disadvantage is that it needs the assumption that $x \geq 0$ (which is fine for some applications).

Algorithm 2

Maintain 3 numbers $w_1 = \sum_{i=1}^{n} x_i$, $w_2 = \sum_{i=1}^{n} i \cdot x_i$, and $w_3 = \sum_{i=1}^{n} x_i \cdot r^i \bmod p$, where $p > n$ is a large prime number and r is a uniform random number from $\{0, 1, 2, \ldots, p-1\}$.

As before, if x is 1-sparse where $x_j \neq 0$, then $w_1 = x_j$ and $w_2 = j \cdot x_j$, and so $(j, x_j) = (w_2/w_1, w_1)$ and hence $w_3 \equiv w_1 \cdot r^{w_2/w_1} \bmod p$ which is used as our test to check whether x is 1-sparse.

Note that if $w_1 = 0$, then either all $x_i = 0$ or there was cancellation, and so in either case x is not 1-sparse. So we assume that $w_1 \neq 0$ in the following argument, which is a common polynomial trick in probabilistic checking.

Claim If x is not 1-sparse, then $w_3 \equiv w_1 \cdot r^{w_2/w_1} \bmod p$ with probability at most n/p.

Proof The equality is equivalent to $\sum_{i=1}^{n} x_i \cdot r^i - w_1 \cdot r^{w_2/w_1} \equiv 0 \bmod p$.

If x is not 1-sparse, then the LHS is a non-zero polynomial of degree at most n.

Note that r is a root of this polynomial for the test to pass.

By the fundamental theorem of algebra, it has at most n roots since its degree is at most n.

Since we chose r uniformly randomly from $\{0, 1, \ldots, p-1\}$, this happens with probability at most n/p. □

Note that we can choose p to be say n^3, so that the failure probability is at most $1/n^2$, while w_3 is still a number of $O(\log n)$ bits.

To answer queries, if $w_3 \neq w_1 \cdot r^{w_2/w_1} \bmod p$ then return "not 1-sparse"; otherwise return $(j, x_j) = (w_2/w_1, w_1)$.

w_3 is still a number of $O(\log n)$ bits.

To answer queries, if $w_3 \neq w_1 \cdot r^{w_2/w_1} \bmod p$ then return "not 1-sparse"; otherwise, return $(j, x_j) = (w_2/w_1, w_1)$.

It has failure probability $n/p \leq 1/n^2$ when x is not 1-sparse but passed the test.

To summarize, we only need to maintain 3 numbers of $O(\log n)$ bits, to answer a query correctly w.p. $\geq 1 - \frac{1}{n^2}$.

---

s-Sparse Recovery (skipped in class)

The idea is to randomly partition the elements in $[n]$ into $100s$ groups / buckets, and hope that each bucket has exactly one non-zero element, so that we can use the 1-sparse recovery algorithm to recover each.

Algorithm

Run $c = \log \frac{s}{\delta}$ copies of the following in parallel

- Choose a random hash function $h : [n] \to [100s]$ from a strongly 2-universal hash family.
- for each bucket $b \in [100s]$,
  - let $x^{(b)}$ be the frequency vector of elements in bucket b, i.e. $\{ i \in [n] \mid h(i) = b \}$.
  - maintain the 1-sparse recovery sketch $sk(x^{(b)})$ of $x^{(b)}$ described in the previous section.
- for each bucket $b \in [100s]$, return the nonzero of $x^{(b)}$ iff $x^{(b)}$ is 1-sparse.

Analysis

Let $S_{nz} := \{ i \mid x_i \neq 0 \}$ be the set of nonzero elements of x.

For each element i in S_{nz}, it will be reported if i is the only nonzero element in its bucket $h(i)$.

By pairwise independence, $\Pr[x^{(h(i))} \text{ is not 1-sparse}] \leq \sum_{j \neq i, j \in S_{nz}} \Pr[h(j) = h(i)] \leq \frac{s}{100s} = \frac{1}{100}$.

So, after running $\log \frac{s}{\delta}$ copies, the element i is not reported with probability at most $\left(\frac{1}{100}\right)^{\log \frac{s}{\delta}} \leq \frac{\delta}{s}$.

By union bound, some nonzero element is not reported with probability at most δ.

Therefore, if x is s-sparse, all nonzero elements of x will be reported with probability at least $1-\delta$.

Checking s-sparsity

The above algorithm will return all non-zeros of x with high probability if x is s-sparse.

If we also want the algorithm to return "non s-sparse" when x is not s-sparse, we can use a similar polynomial trick to detect it with high probability.

Maintain $w = \sum_{i=1}^{n} x_i \cdot r^i \bmod p$ where $p > n$ is a large prime and r is a uniform random element in $\{0, \ldots, p-1\}$.

Also maintain $\tilde{w} = \sum_{i \in S_{report}} x_i \cdot r^i \bmod p$, where S_{report} is the set of nonzero elements reported by the algorithm.

Then, using the same argument, if x is not s-sparse, then $w = \tilde{w}$ with probability at most $n/p \leq \delta$ if we set $p \geq n/\delta$. We leave the details to the reader.

Space: We run a total of $c \cdot 100s = 100s \log \frac{s}{\delta}$ copies of the 1-sparse recovery algorithm, where each copy maintain 3 numbers of $O(\log n)$ bits.

So, we get a sketch $sk(x) \in \mathbb{Z}^{O(s \log \frac{s}{\delta})}$, and check that $sk(x)$ is a linear sketch.

There are $\log \frac{s}{\delta}$ hash functions, each require $O(1)$ numbers of $O(\log n)$ bits to store.

ℓ_0- Sampling

We will also use 1-sparse recovery as a building block to construct a sketch for ℓ_0-sampling.

The high level idea is as follows.

Suppose we know $\|x\|_0$ (which we don't). Then we can sample each element in $[n]$ with probability $1/\|x\|_0$.

Then, as we seen before in L02 for isolating cuts, exactly one non-zero element of x is sampled, and so we can recover it using the 1-sparse recovery algorithm. Note that the retured element is uniformly random.

But we don't know $\|x\|_0$. The fix is to guess $\|x\|_0$ by trying all powers of 2.

Another issue is that we cannot afford to sample each element independently as it requires too much space to remember the sampled subset for the update operations. The fix is to use a hash function.

Algorithm

Run $c = 1000 \log n$ copies of the following in parallel.

- Choose a random hash function $h : [n] \to [n]$ from a strongly 2-universal hash family.
- for $0 \le k \le \log n$, let $T_k := \{ i \in [n] \mid h(i) \le n/2^k \}$.
- for $0 \le k \le \log n$, run 1-sparse recovery $sk^1(x_{T_k})$ of x_{T_k}.
- for $0 \le k \le \log n$, return the non-zero entry of x_{T_k} if $\|x_{T_k}\|_0 = 1$.

  (If the algorithm returns several nonzeros, just return one of them.)

Space: There are $c = 1000 \log n$ copies. In each copy, there are $\log n$ many levels.

In each level, the 1-sparse recovery sketch needs only 3 numbers.

So, the sketch requires $O(\log^2 n)$ numbers. It is a linear sketch as 1-sparse recovery is linear.

Additionally, we need to store $O(\log n)$ hash functions, for a total of $O(\log n)$ numbers.

Correctness: If we sample each element (fully) independently, then the following claim is easy to prove as in L02.

Here we need to work more carefully since we only use a pairwise independent hash function.

Claim Suppose $2^{k-2} \le \|x\|_0 \le 2^{k-1}$. Let $T_k = \{ i \in [n] \mid h(i) \le n/2^k \}$. Then $\Pr[\|x_{T_k}\|_0 = 1] \ge 1/8$.

proof $\Pr[\|x_{T_k}\|_0 = 1] = \sum_{i \in S_{nz}} \Pr[i \in T_k \text{ and } j \notin T_k \text{ for all } j \in S_{nz} \setminus \{i\}]$

proof $\Pr[\|x_{T_k}\|_0 = 1] = \sum_{i \in S_{nz}} \Pr[i \in T_k \text{ and } j \notin T_k \text{ for all } j \in S_{nz} \setminus \{i\}]$

$= \sum_{i \in S_{nz}} \Pr[i \in T_k] \cdot \left(1 - \Pr[j \in T_k \text{ for some } j \in S_{nz} \setminus \{i\} \mid i \in T_k]\right)$

$\geq \sum_{i \in S_{nz}} \Pr[i \in T_k] \cdot \left(1 - \sum_{j \in S_{nz} \setminus \{i\}} \Pr[j \in T_k \mid i \in T_k]\right)$ // union bound

$= \sum_{i \in S_{nz}} \Pr[h(i) \leq n/2^k] \cdot \left(1 - \sum_{j \in S_{nz} \setminus \{i\}} \Pr[h(j) \leq n/2^k]\right)$ // pairwise independence

$= \sum_{i \in S_{nz}} \frac{1}{2^k} \cdot \left(1 - \frac{|S_{nz}| - 1}{2^k}\right)$

$\geq \sum_{i \in S_{nz}} \frac{1}{2^k} \left(\frac{1}{2}\right)$ // $|S_{nz}| \leq 2^{k-1}$

$\geq \frac{1}{8}$ // $|S_{nz}| > 2^{k-2}$ □

Graph Sketching

We will use the ℓ_0-sampling sketches to design some graph streaming algorithms.

The approach is to store the sketches of the graph and use them to answer queries to solve some graph problems.

The graph problems that we will solve are about cuts.

The following matrix representation of a graph is crucial for solving cut problems.

Incidence Matrix Let $G = (V, E)$ be a simple undirected graph.

The incidence matrix A_G is an $n \times \binom{n}{2}$ matrix, where each row is indexed by a vertex and each column is indexed by a potential edge (i.e. all possible pairs of vertices).

For a column indexed by (i,j), if $(i,j) \notin E$, then it is a zero column,

while if $(i,j) \in E$, then it is a column with $+1$ in the i-th entry, -1 in the j-th entry, and zero otherwise.

Each row is an $\binom{n}{2}$-dimensional vector, and each column is an n-dimensional vector with at most 2 nonzer

The important property of the incidence matrix is that we can read the edges in a cut $C \subseteq V$ by adding the rows in C.

Claim Let a_i be the i-th row of A_G. For any cut $C \subseteq V$, let $a_C := \sum_{i \in C} a_i$.

Then $a_C(j,k)$ is nonzero if and only if (j,k) is an edge in $\delta(C)$ (i.e. crossing the cut C).

Proof If the edge (j,k) has both endpoints in C, then $a_C(j,k) = 0$ because of cancellation.

If the edge (j,k) has both endpoints in $V-C$, then $a_C(j,k) = 0$ obviously.

If the edge (j,k) has one endpoint in C and one endpoint in $V-C$ (i.e. $(j,k) \in \delta(C)$),

then $a_C(j,k)$ is either $+1$ or -1. □

For graph streaming algorithms, we would like to use as little space as possible to store the graph.

while still be able to answer useful queries for solving problems.

The main idea is to only maintain a sketch of each row a_i for ℓ_0-sampling. So, instead of storing an $\binom{n}{2}$-dimensional vector for each row, we just store and maintain an ℓ_0-sampling sketch $sk^0(a_i)$ of a_i of polylog(n)-dimension for computation. So, the total storage is $O(n \cdot \text{polylog}(n))$.

Recall the important property that the sketches described in previous sections are all <u>linear</u> sketches. So, there is some (implicit) matrix M such that $sk(x) = M \cdot x$. where the matrix M is defined by e.g. hash functions using much less space and still $sk(x)$ can be efficiently computed.

(The reader is encouraged to write this matrix explicitly for the linear sketches that we've seen.)

One advantage of maintaining a linear sketch is that it is easy to update, if $x \leftarrow x + \Delta$ then $sk(x+\Delta) = sk(x) + sk(\Delta)$ so that we only need to compute $sk(\Delta)$ and add to $sk(x)$.

<u>Finding an edge in a cut</u>

In applications that we will see, we would like to compute a_C and use it to find some edge in $\delta(C)$.

To do so, we maintain the ℓ_0-sampling sketch $sk^0(a_i)$ for each vertex $i \in V$, and use them to compute $sk^0(a_C) = Ma_C = M(\sum_{i \in C} a_i) = \sum_{i \in C} sk^0(a_i)$.

Note that we use the same hash functions for each vertex $i \in V$, so that their sketches are based on the same linear transformation M so that the above identity holds.

Since we maintain an ℓ_0-sampling sketch $sk^0(a_C)$ of a_C, we can use it to return a random non-zero entry of a_C, which is a random edge in $\delta(C)$.

This is the main building block of the streaming algorithms for graph cut problems that we will see.

Graph Streaming Algorithms

The setup of graph streaming algorithms is that we start from an empty graph with n vertices $\{1, 2, \ldots, n\}$, and there is a dynamic stream which is a sequence of edge insertion or edge deletion updates.

We assume that an edge deletion occurs only when the edge exists, i.e. no "negative" edge in the graph.

At any point of time, we may need to answer queries about the current graph, e.g. whether it is connected, what is the minimum cut value, etc.

We don't know the updates nor the queries in advance.

If we store the whole graph, then there is nothing special about this setting.

The challenge is to use as little space as possible, so that we can still update efficiently and answer the queries (approximately) correctly with high probability.

It is crucial that we allow the algorithms to make mistakes in this setting, as it is possible to prove that there are deterministic algorithms that only need $o(n^2)$ space.

As discussed in the previous section, we will use the incidence matrix of the graph, but only maintain a sketch $sk(a_i)$ of each row of the matrix that corresponds to each vertex i in the graph. We will use the same random bits for each row (e.g. same hash functions) so that the sketches satisfy linearity $sk(a_C) = \sum_{i \in S} sk(a_i)$ as discussed in the previous section.

When an edge (i,j) comes in, we update $sk(a_i)$ and $sk(a_j)$ using the algorithm depending on which type of sketch.

When a query comes in, we need to compute the answer using sketches $sk(a_1), \ldots, sk(a_n)$ only.

We now "sketch" a few interesting streaming algorithms for some graph cut problems.

Finding a spanning forest

We adapt the classical Boruvka's algorithm for finding spanning forests.

In this algorithm, we maintain a set of supernode which represents a subset of vertices in the graph. The intention is the each supernode is a subset of vertices that belong to the same component in the graph.

Initially, there are n supernodes $C_1, C_2, \ldots, C_n$, each is simply a vertex i in the graph.

In each round, for each supernode C_i, we find an edge $jk \in \delta(C_i)$. If no such edges exist for a supernode C_i, then we know that C_i is a connected component in the graph and we call it "inactive" from now on.

We store all these edges, and collapse the supernodes that are connected by these edges into one supernode.

Note that the number of "active" supernodes after each round is at decreased by a factor of 2, and so this algorithm stops in at most $\log_2 n$ rounds.

Then, we can find a spanning forest using the edges that we stored in these rounds.

It should not be difficult to see how we can use the sketches to execute this algorithm.

In each round, we use l_0-sampling sketches $sk^0(a_1), \ldots, sk^0(a_n)$ to compute $sk^0(a_C) = \sum_{i \in C} sk^0(a_i)$ for each supernode C, and then use $sk^0(a_C)$ to return an edge jk in $\delta(C)$.

To avoid dependency between the rounds (i.e., the input to the next round depends on the output of the current round), we use a new set of sketches $sk^0(a_1), \ldots, sk^0(a_n)$ in each round, i.e. in the l_0-sampling sketch earlier, we use independent hash functions to compute the sketches in each round.

So, in total, we use $O(n \log n)$ sketches of the form $sk^0(a_i)$, where each such sketch requires $O(\log^2 n)$ numbers of $O(\log_2 n)$ bits by the result for l_0-sampling. Thus, total space required is $\tilde{O}(n)$ space.

k-Edge-Connectivity

The spanning forest algorithm can be used as a building block to solve the k-edge-connectivity problem, which is to determine whether the min-cut value of an unweighted undirected graph is at least k or not.

The main idea is based on the following combinatorial lemma (which will be explained in class).

Lemma Given G and $k \geq 0$, let F_i be a spanning forest of $G \setminus (F_1 \cup F_2 \cup \dots \cup F_{i-1})$.

Then G is k-edge-connected if and only if $H = F_1 \cup F_2 \cup \dots \cup F_k$ is k-edge-connected.

First, we construct the sketches needed for k independent instantiations $\mathcal{I}_1, \dots, \mathcal{I}_k$ of the spanning forest algorithm.

Then, use $\mathcal{I}_1$ to compute a spanning forest F_1 of G.

Then, update $\mathcal{I}_2$ by removing all edges in F_1 from G, and then compute a spanning forest of $G - F_1$.

And so on, until we use $\mathcal{I}_k - (F_1 \cup F_2 \cup \dots \cup F_{k-1})$ to compute a spanning forest of $G - (F_1 \cup \dots \cup F_{k-1})$.

Then, we check whether $H = F_1 \cup \dots \cup F_k$ is k-edge-connected (using any reasonable algorithm).

The space complexity is $\tilde{O}(n \cdot k)$, so it only works well when k is small.

Approximate Minimum Cut ("Sketch")

The k-edge-connectivity algorithm can in turn be used as a building block for approximating minimum cut.

The new ingredient, as you may have guessed, is karger's graph sparsification result in L04.

If the min-cut value is $O(\log n)$, then we can use the k-edge-connectivity algorithm to compute an exact minimum cut using $\tilde{O}(n)$ space.

If the min-cut value is much larger than $\log n$, then the idea is to use karger's uniform sampling algorithm to sparsify the graph G s.t. the minimum cut in the sparsified graph H has $O(\log n)$ edges, and use the scaled-up min-cut value as a good approximation (i.e. min-cut$(H)/p$ where p is sampling prob).

To find out the correct sampling probability, we can try all powers of 2, i.e. $p = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$

So we run $O(\log n)$ copies of the k-edge-connectivity algorithm, each with a different sampling p.

It is not difficult to show that the largest p so that the min-cut value in the sparsified graph H becomes $O(\frac{1}{\varepsilon^2} \cdot \log n)$, then min-cut$(H)/p$ is an $(1+\varepsilon)$-approximation to the min-cut value.

So, in total, we need $\tilde{O}(\frac{1}{\varepsilon^2} \cdot n)$ space for approximating minimum cuts.

A technical issue is how to do the random sampling.

A natural idea is to use hash functions, but the problem is that bounded-independence is not enough for the analysis to show that the sparsified graph is an $(1 \pm \varepsilon)$-cut approximator.

The fix is to use pseudorandom generator for space-bounded computation, which is slightly outside the scope of the course.

References

This lecture is heavily based on Lecture 6.1 and 6.2 of "Introduction of algorithms" by Thatchaphol Saranurak (Michigan 2023), and Lecture 5 of "Graph Streaming Algorithms and Lower Bounds" by Sepehr Assadi (Rutgers 2020).

Some original references are:

- Cormode and Firmani, A unifying framework for l_0-sampling algorithms, 2014.
- Ahn, Guha, McGregor, Analyzing graph structure via linear measurements, 2012.

More recently, these ideas are used to design fast data structures for graph algorithms.