Streaming Algorithms for Independent Sets

Bjarni V. Halldórsson[†] Magnús M. Halldórsson[‡] Elena Losievskaja[‡§]
Mario Szegedy[¶]

April 28, 2010

Abstract

We find "combinatorially optimal" (guaranteed by the degree-sequence alone) independent sets for graphs and hypergraps in linear space in the semi-streaming model.

We also propose a new *output-efficient* streaming model, that is more restrictive than semi-streaming ($n \cdot \log^{O(1)} n$ space) but more flexible than classic streaming ($\log^{O(1)} n$ space). The algorithms in this model work in poly-logarithmic space, like in the case of the classical streaming model, but they can access and update the output buffer, treating it as an extra piece of memory.

Our results form the first treatment of the classic IS problem in the streaming setting.

1 Introduction

In this paper we consider streaming algorithms for the classic independent set problem on graphs and hypergraphs. As input, we are presented with a (hyper)graph edge by edge, and we have to output a large set of vertices that contains no (full) edges. For graphs, a theorem of Paul Turan guarantees an independent set of size $n/(d+1)$, where d is the average degree of the graph. If the entire degree-sequence, $d_1, \ldots, d_n$ of the graph is available, then $\sum_{i=1}^{n} \frac{1}{d_i+1}$ is a stronger lower bound for the maximum independent set size (in this paper n will always denote the number of nodes of the input graph). The formula has a generalization for hypergraphs as well (see Section 1.2). This "combinatorially optimal" output size is what we will require of our algorithms.

In the *streaming model* [13], the data is presented sequentially in the form of a data stream, one item at a time, and the working space (the memory used by the algorithm) is significantly less than the size of the data stream. The motivation for the streaming model comes from practical applications of managing massive data sets such as, e.g., real-time network traffic, online auctions, and telephone call records. These data sets are huge and arrive at very high rate, making it impossible to store the entire input and forcing quick decisions on each data item. However, most graph problems are impossible to solve within the polylogarithmic space bound that the traditional streaming model offers (a rare exception is an algorithm for the problem of counting triangles in a graph [5]).

[†]School of Science and Engineering, Reykjavik University, 101 Reykjavik, Iceland, e-mail: `bjarnivh@ru.is`

[‡]School of Computer Science, Reykjavík University, 101 Reykjavik, Iceland, e-mail: {`mmh@ru.is`,`ellossie@gmail.com`}. Research supported by grant 7000921 of the Iceland Research Fund

[§]Current address: Icelandic Heart Association, Holtasmari 1, 201 Kopavogur, Iceland.

[¶]Dept. of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, New Jersey, USA, e-mail: `szegedy@cs.rutgers.edu`Supported by NSF grant EMT-0523866.

This observation led to the introduction of the *semi-streaming* model [13], where space for n-vertex graphs is restricted to $n \log^{O(1)} n$. The algorithms then have enough space for some information for each vertex, but not enough to store the whole graph. A number of graph problems have been studied in the semi-streaming model, including bipartite matching (weighted and unweighted cases) [10, 9], diameter and shortest paths [10, 11], min-cut [1], and graph spanners [11]. The independent set (IS) problem, to our best knowledge, has not been studied in the model before.

In the case of IS, the $n \log^{O(1)} n$-bound of semi-streaming seems overly generous, but we cannot reasonably expect a sublinear bound, given the need to store the problem solution. Instead, we suggest that the focus be placed on the amount of *extra space* required, i.e. in addition to that required for storing the solution.

All the algorithms we consider have the common feature that they maintain a feasible solution at all times. Also, decisions made on edges are irreversible: once a node is ejected from the solution, it never enters it again. This defines a new *online streaming* model. It is closely related to *preemptive online* algorithms, considered recently by Epstein et al.[9] in a streaming context for weighted matching. The difference is that in our problem, we can view the whole vertex set as belonging to the initial solution, thus the solution of any algorithm is *monotonously non-increasing* with time. There have been few works on online graph problems that allow for one-way changes. A rare example is a recent work of [8] that can be viewed as dealing with maintaining a strong independent set of a hypergraph when edges arrive in a stream.

To contrast, in the classical *online* version of the IS problem [12, 2], vertices arrive one by one, along with all incident edges to previous vertices. The online algorithm must then determine once and for all whether the node is to be included in the constructed feasible independent set solution. This problem is very hard to approximate [12, 2]; e.g., a competitive factor of $n-1$ is best possible for deterministic algorithm, even when restricted to trees. However, bounded-degree graphs are comparatively easy, since a factor of Δ is trivial for a deterministic greedy algorithm.

By focusing on the space requirements and the way the working space can be used, we seek to gain a deeper understanding of the way independent sets can be computed. We generally try to avoid any prior assumptions such as knowing the set or the number of vertices in advance, and we are willing to pay a well-calculated price for this in terms of extra space.

1.1 Our Contribution

We design space-efficient semi-streaming algorithms for IS. Our starting point is algorithm RandomOffLine of Shachnai and Srinivasan [14] (see Fig. 2). We modify this algorithm to achieve:

- $O(n \log r)$ space, instead of $n \log n$, where r is the cardinality of the largest hyperedge (Algorithm RandomPartialPermute). Also, n does not need to be known for the algorithm in advance.

- $O(n)$ space (Algorithm RandomMap; here n needs to be known in advance, and the constants are somewhat compromised).

We also analyse the situation, where we are allowed to keep only a single bit (!) per node.

Finally, we explore new models of semi-streaming and argue upper and lower bounds. In the on-line semi-streaming model output-changes can go only in one direction and a feasible solution must be maintained at all times. Also, *memoryless* algorithms are allowed only logarithmic *extra* space in addition to the bits required to store the solution.

1.2 Definitions

Given a hypergraph $H = (V, E)$, let n and m be the number of vertices and edges in H, respectively. We assume that H is a *simple* hypergraph, i.e. no edge is a proper subset of another edge. An *independent set* I in H is a subset of vertices that contains no edge of H. If I is independent, then $V \setminus I$ is said to be a *hitting set*.

A hypergraph is *r-uniform* if all edges have the same cardinality r. Graphs are exactly the 2-uniform hypergraphs. For graphs and hypergraphs the *degree* $d(v)$ of a vertex v is the number of edges incident on v. We denote by Δ and d the maximum and the average degree, respectively. For us, a much better measure for the degree of the vertex v of H is $1/p$, where p is the solution of $\sum_{e\ni v} p^{|e|-1} = 1$. This we call the *efficient degree* of v, and we denote it by $d^*(v)$. Also, define $i(H) = \sum_v \frac{1}{d^*(v)}$. For intuition, note that for a d-regular k-uniform hypergraph H, $i(H) = n/\sqrt[k-1]{d}$. Let $\alpha(H)$ be the maximum independent set size of H. What makes $i(H)$ interesting for us is that $\alpha(H) = \Omega(i(H))$. In fact, $\Omega(i(H))$ is the strongest lower bound for $\alpha(H)$ we can obtain from the degree-sequence alone. Let IS denote the problem of finding an independent set in hypergraphs of size $\Omega(i(H))$.

We can generalize all of the definitions for weighted (hyper)graphs. A vertex-weighted hypergraph has a non-negative weight function on its vertices. The notions of average degree etc. carry over to the weighted case in a natural way, for instance $i(H)$ becomes $\sum_v \frac{w(v)}{d^*(v)}$, where w is the weight function. Most of our results will carry over to the weighted case with obvious modifications.

We also note that our algorithms will be such that if vertices with degree 0 should appear in H, then they will be automatically included in the independent set that the algorithm outputs.

We assume that the vertices are labelled $0, \ldots, n-1$. This assumption can be voided by simply maintaining a lookup table, but the storage requirements for such lookup are beyond the scope of our considerations here.

For this article, the most precious resource is space, and we model and regard memory as a linear array of bits with direct access (as in standard C programming).

2 A Basic Algorithm

All of our algorithms will be based on the well known observation, that in a sparse graph random sets of the right size are nearly independent. The algorithm we present in Fig. 1 is crucial to the analysis of the subsequent algorithms in later sections.

Remark. On a random set of size pn we shall mean either of the two things: 1. We select a subset of $V = [n]$ of size pn uniformly and randomly. 2. We create a random subset X of V by the procedure that selects all nodes in V randomly and independently with probability p. While we prove Lemma 2.1 only for case 2, our applications will use lemma for case 1, where it also holds.

We now analyze $\textsc{Basic}[p,\ H]$.

Lemma 2.1 *Let $H(V, E)$ be a hypergraph and let A be the set of nodes of H with efficient degree $\leq \frac{1}{2p}$. Then $\textsc{Basic}[p,\ H]$ will return an independent set I such that $\mathbb{E}[|A \cap I|] \geq p|A|/2$.*

Proof: For a node v, let χ_v be the random variable that takes value 1 if the node is selected into I, and 0 otherwise. Then $|A \cap I| = \sum_{v\in A} \chi_v$. To estimate the expectation of χ_v from below, notice that if v is in X, but for every edge e incident on v we have $e \not\subseteq X$, then $v \in I$. The probability that $v \in X$ is p. The conditional probability that for an e with $v \in e$ we have that

```
ALGORITHM BASIC[$p$, $H$]
 Input: a hypergraph $H(V, E)$, probability value $p$

 $S \leftarrow \emptyset$
 Let $X$ be a random subset of $V$ of size $pn$.
 For each edge $e \in E(H)$ do
     If $e \not\subseteq X$, let $v$ be any vertex in $e \setminus X$;
     otherwise, let $v$ be any vertex of $e$.
     $S \leftarrow S \cup \{v\}$
 Output $I = V \backslash S$
```

Figure 1: Algorithm BASIC is crucial for most of our performance analysis

$e \not\subseteq X$ is $1 - p^{|e|-1}$. If the edges incident on v would have only vertex v in common, and were otherwise disjoint, then this would give us the probability

$$p \prod_{e:\, v \in e} \left(1 - p^{|e|-1}\right) \geq p \left(1 - \sum_{e:\, v \in e} p^{|e|-1}\right) \geq p/2$$

for the desired event, where the last inequality follows from $\sum_{e:\, v \in e} (2p)^{|e|-1} \leq 1$, since the efficient degree of any $v \in A$ by our assumption is at least $1/(2p)$. When the (hyper)edges that are incident on v arbitrarily intersect, we use the FKG-inequality as in [14] towards getting the exact same estimate. The latter implies that the correlations that arise by letting the edges overlap will make the event that none of them is contained in X more likely (we omit the details).

The lemma now follows from the additivity of expectation. ■

3 Permutation-based Algorithms

The idea of permutation-based algorithms is to randomly permute vertices and use this random permutation to decide which vertices to include in the independent set. Given a set of vertices V, the randomized algorithm RANDOMOFFLINE (see Fig. 2) creates a permutation π on the given set V of vertices. The last vertex of each edge is added to a set S. The set S forms a hitting set and $V \backslash S$ an independent set.

```
ALGORITHM RANDOMOFFLINE
 Input: a hypergraph $H(V, E)$

 $S \leftarrow \emptyset$
 Let $\pi$ be a random permutation of the vertices in $V$
 For each edge $e \in E(H)$ do
     Let $v$ be the last vertex in $e$ with respect to $\pi$
     $S \leftarrow S \cup \{v\}$
 Output $I = V \backslash S$
```

Figure 2: The off-line algorithm RANDOMOFFLINE

Shachnai and Srinivasan [14] proved the following performance bounds for RANDOMOFFLINE on hypergraphs. An elegant proof for graphs is featured in the book of Alon and Spencer [4].

Theorem 3.1 ([14]) *Given a hypergraph H, the off-line algorithm* RandomOffLine *finds an independent set of expected weight $\Omega(i(H))$.*

A simple heuristic improvement to the algorithm is to add the last vertex of an edge to S only if that edge doesn't already have a representative in the set, i.e., if $e \cap S \neq \emptyset$. Since the addition of this condition never decreases the solution size, the performance claims continue to hold.

It is immediately clear that RandomOffLine is actually a streaming algorithm, since it treats the edges in whichever given order. We can even avoid the assumption that the algorithm knows the number n of vertices in advance. For that, we construct a random permutation π on-the-fly by randomly inserting each new vertex in the ordering of the previous vertices.

We can significantly reduce the space by constructing a partial random permutation instead of a complete random permutation. Each vertex v is associated with a finite sequence s_v of random bits, where each bit in s_v is independently set to 0 or 1 with equal probability. A partial random permutation of the vertex set V is then a lexicographic order of the corresponding bit-sequences $\{s_v\}_{v \in V}$. Consider an edge $e \in E$. Let $\{s_v | v \in e\}$ be the set of bit-sequences associated with the vertices in e. For vertices $u, v \in e$, we write $u \succ v$ if s_u follows s_v in lexicographical order. We define the vertex $u \in e$ to be the *last* in e, if s_u is lexicographically last in the set $\{s_v | v \in e\}$. The idea is that for each vertex $v \in e$ we use the minimum number of bits required to determine if s_v is the last in $\{s_v | v \in e\}$. In other words, we just need to determine which vertex in e is the last, and the relative order of other vertices in e is not important. The formal description of this algorithm RandomPartialPermute is given in Fig. 3. Let $s_v[j]$ be the j-th bit in s_v.

```
Algorithm RandomPartialPermute
 Input: a stream $E$ of edges

 $V \leftarrow S \leftarrow \emptyset$
 For each edge $e$ in the stream $E$ do
     For each vertex $v \in e$ such that $v \notin V$ do
         $V \leftarrow V \cup \{v\}$
         $s_v \leftarrow \emptyset$
     $U \leftarrow \{e\}$
     $j \leftarrow 1$
     While $|U| \neq 1$ do
         For each $v \in U$ such that $s_v[j]$ is not defined do
             $s_v[j] = 1$ with probability $\frac{1}{2}$, otherwise $s_v[j] = 0$
         If $\exists v \in U$ such that $s_v[j] = 1$
             $U \leftarrow U \backslash \{v \in U \mid s_v[j] = 0\}$
         $j \leftarrow j + 1$
     $S \leftarrow S \cup U$
 Output $I = V \backslash S$
```

Figure 3: The algorithm RandomPartialPermute

As stated, the algorithm RandomPartialPermute is not fully implemented. Specifically, it remains to organize the bits stored into a structure that can be easily accessed. Various approaches are possible that all complicate the picture. We will instead leave the idea in this partially developed state.

Theorem 3.2 RandomPartialPermute *finds an independent set of expected weight $\Omega(i(H))$*

using expected $O(n \log r)$ space and $O(r)$ time to process each edge.

Proof: First, we show that the ordered set $S = \{s_v | v \in V\}$ forms a partial permutation of V. We note that a random permutation can be created by assigning each vertex a random number drawn from the uniform distribution on $[0, 1)$ and asserting that u follows v in π, if the random number assigned to u is greater than the random number assigned to v. A uniform random number drawn from $[0, 1)$ can be viewed as an infinite sequence of unbiased random bits. RandomPartialPermute creates such a bit-sequence with only as many bits created as needed to determine if s_v is lexicographically last in $\{s_v | v \in e\}$ for every edge e such that $v \in e$. Therefore, the set $\{s_v | v \in V\}$ forms a partial permutation of V and we can apply the same argument as in the proof of Theorem 3.1 to show that RandomPartialPermute finds an independent set of expected weight $\Omega(i(H))$.

In the remainder we show that the algorithm uses $O(n \log r)$ space to store the set $\{s_v\}$ of bit-sequences. Given a vertex v, we say that we *open* the j-th bit in s_v, if the algorithm assigns $s_v[j]$ to either 0 or 1.

Consider an edge e incident on v. Let $s_{v,e}$ be the bit-sequence s_v at the time e appears in the stream and let $u \succ v$ if $s_{u,e} > s_{v,e}$. Let $U_{\succ v}(e) = \{u \in e | u \succ v\}$, $U_{\prec v}(e) = \{u \in e | v \succ u\}$ and $U_{=v}(e) = \{e\} \backslash (U_{\succ v}(e) \cup U_{\prec v}(e))$. We need to open more bits in $s_{v,e}$ only if $U_{\succ v}(e) = \emptyset$ and $U_{=v}(e) \neq \emptyset$, because in this case the vertices in $U_{=v}(e)$ have the highest bit-sequences and these bit-sequences are exactly the same as $s_{v,e}$. In this case we say that e is *problematic for v*. In any other case, the opened bits in $s_{v,e}$ are sufficient to decide which vertex covers e, namely e is covered either by a vertex $u \in U_{\succ v}(e)$ if $U_{\succ v}(e) \neq \emptyset$ or by the vertex v if $U_{\succ v}(e) = \emptyset$ and $U_{=v}(e) = \emptyset$.

To simplify the analysis we will open bits in batches, $3 \log r$ bits are opened initially and immediately following the resolution of a problematic edge, and further as many bits are opened as are needed to resolve a problematic edge.

Consider an edge e problematic for v. We compute an upper bound on the expected number of vertices that have the same bit-sequence as v and condition the computation of this expectation over all the values that v can take. Each time we encounter a problematic edge we are guarenteed to have $3 \log r$ bits which may be considered to be random. The bit-sequences that the other vertices in e take are conditioned on being less than or equal to the bit-sequence for v. Then, the expected number of vertices that have the same bit-sequence as v in e at the $3 \log r$ bits under consideration can be determined by: summing over all r^3 possible values that the bit-sequence for v can take and multiplying the probability that v takes the value of a given bit-sequence, $\frac{1}{r^3}$, with the expected number of other nodes in e that take the same bit-sequence value. The expected number of nodes in e taking the same bit-sequence value as v is bounded by $\frac{(r-1)}{i+1}$, where i is the bit-sequence value of v (As $| e - \{v\} | \leq r - 1$ and $i+1$ is the number of bit-sequences $\leq i$). We get an expression

$$|U_{=v}(e)| \leq \sum_{i=0}^{r^3} \frac{1}{r^3}(r-1)\frac{1}{i+1} \leq \frac{(3 \log r + 1)\frac{r-1}{r}}{r^2} \leq \frac{1}{r} \ .$$

Each time we encounter a problematic edge e, we open $3 \log r$ bits and then as many bits as needed to determine which of the vertices in $U_{=v}(e)$ has the highest bit-sequence, in expectation $\log(1 + \frac{1}{r})$ bits. In total we open in expectation

$$\sum_{i=0}^{\infty} \left(3 \log r + \log(1 + \frac{1}{r})\right) \left(\frac{1}{r}\right)^i \leq 6 \log r$$

bits. ■

3.1 Linear Space Algorithm

Interestingly, we can run algorithm Basic "in parallel," for many different p's at the same time in the same *sequential* algorithm! This happens in the case of algorithm RandomOffLine. For a random permutation π, define

$$X_k = \{\text{first } k \text{ elements of } \pi\} .$$

Now X_k is a random set with size pn, where $p = k/n$. Notice that upon running RandomOffLine we also run Basic for X_k for $k = 1, \ldots, n$ in parallel. Indeed, it holds that when a hyperedge e is processed, we add the last vertex, v, of e to S. Thus, unless $e \subseteq X_k$, vertex v is not in X_k. Using this property, we see that for every vertex v the expectation of χ_v of Lemma 2.1 is at least $\frac{1}{4(d^*(v)+1)}$, just by picking X_k for the node with $k = n/d^*$, and applying the same argument to bound $\mathbb{E}[\chi_v]$ from below, as in Lemma 2.1. Theorem 3.1 (aside from a constant factor) now follows easily from the additivity of expectation.

We now create a new, more efficient algorithm, RandomMap from the observation that the above argument goes through even when we only use the properties of X_k for $k = 1, 2, 4, 8, \ldots$. Indeed, if v has efficient degree d^* then choose $\frac{n}{2d^*} \leq k \leq \frac{n}{d^*}$. Then, we get that χ_v of Lemma 2.1 is at least $\frac{1}{8(d^*(v)+1)}$.

```
Algorithm RandomMap
 Input: a hypergraph $H(V, E)$

 $S \leftarrow \emptyset$
 Let $\rho$ be a random map $V \longrightarrow [\log n]$ described below.
 For each edge $e \in E(H)$ do
     Let $v$ be the vertex in $e$ with the largest image in $\rho$
       (with conflicts resolved arbitrarily)
     $S \leftarrow S \cup \{v\}$
 Output $I = V \backslash S$
```

Figure 4: Algorithm RandomMap

To provide the X_k, for all powers of two, we do not need to create a permutation of the nodes. A map $\rho : V \longrightarrow [\log n]$ suffices, where the probability of $\rho(v) = i$ is $1/2^i$ (to make the total probability equal to one, we set the probability of $\rho(v) = \log n$ to be $2/n$ instead of $1/n$).

Our new algorithm runs in the same way as RandomOffLine, except that π is replaced by ρ, and when conflicts (i.e., when the smallest element is not unique) are resolved arbitrarily.

We now describe how to store and access ρ using small space. Since the domain of ρ has size $\log n$, there is a straightforward random access implementation that uses only space $n \log \log n$ by storing $\rho(i)$ in the memory segment $[(i-1)\lceil \log \log n \rceil + 1, i\lceil \log \log n \rceil]$. We can use the space even more efficiently, however. Notice that $\sum_{i=1}^{n} \log \rho(i) = O(n)$. Thus, the sequence

$$\rho(1) \ \$ \ \rho(2) \ \$ \ldots \$ \ \rho(n)$$

has bit-length linear in n. But this does not facilitate the binary search for $\rho(i)$, since there is no indication in the above sequence which ρ value we are reading. And since the $\rho(i)$'s have random bit length, we cannot directly access the address of $\rho(i)$ either. One of several possible solutions is to insert markers in $n/\log n$ places to guide our search. We put a marker before $\rho(1)$, $\rho(1+\log n)$, $\rho(1+2\log n)$, and so on. Each marker has bit-length $\log n$, and tells which ρ-value comes immediately after it. The new sequence with the markers obviously still occupies linear space. It is easy to see that now we can implement a binary search where each step of the search involves finding the closest marker and to decide whether to go left or right. Then after each step the search interval is halved (in terms of bit-length). At the final step a short sequential search leads to the desired $\rho(i)$. The worst case running time is $O((\log n)^3)$. There are various ways to make this algorithm more time-efficient.

4 Online Streaming Algorithms

All the algorithms considered in this paper have the following two properties: (a) they maintain a feasible solution I at all times, and (b) rejection decisions (i.e., the removal of a node from I, or alternatively addition to S) are irrevocable. We refer to such algorithms as *online streaming* algorithms.

In this section, we study the power of this model in the deterministic case. We restrict our attention here to the case of graphs.

Deterministic algorithms: The next result shows that no deterministic algorithm can attain a performance ratio in terms of d alone, nor a ratio of $2^{o(\Delta)}$.

Theorem 4.1 *The performance ratio of any deterministic algorithm in the online streaming model is $\Omega(n)$. This holds even for trees of maximum degree $\log n$, giving a bound of $\Omega(2^{\Delta})$. It also holds even if the algorithm is allowed to use arbitrary extra space.*

Proof: Assume that $n = 2^k$ is a power of 2. Let A be any deterministic algorithm.

We maintain the invariant that the independent set selected by A contains at most one node in each connected component. We join the n vertices together into a single tree in k rounds. In round i, for $i = 1, 2, \ldots, k$, $n/2^i$ edges are presented. Each edge connects together two components; in either component, we choose as endpoint the node that is currently in A's solution, if there is one, and otherwise use any node in the component. This ensures that the algorithm cannot keep both vertices in its solution, maintaining the invariant.

In the end, the resulting graph is a tree of maximum degree at most k, and A's solution contains at most one node. ■

When allowing additional space, we can match the previous lower bound in terms of Δ.

Theorem 4.2 *There is a deterministic algorithm in the online streaming model with a performance ratio of $O(2^{\Delta})$.*

Proof: We consider an algorithm that maintains additional information in the form of a counter c_v for each node v, initialized as zero.

When an edge arrives between two nodes in the current solution I, we compare the counters of the nodes. The node whose counter is smaller, breaking symmetry arbitrarily, is then removed from the current solution I. The counter of the other node, e.g. u, is then increased. We then say that u *eliminated* v. We say that a node u is *responsible* for a vertex x if u eliminated x, or, inductively, if u eliminated a node that was responsible for x.

Let $R(k)$ denote the maximum, for any node v with $c_v = k$, of the number of nodes for which v is responsible. We claim that $R(k) \leq 2^k - 1$. It then follows that the size of I is at least $n/2^{\Delta}$, since c_v is at most the degree of v. For the base case $R(0) = 0$, since the node never eliminated another vertex. Assume now that $R(t) \leq 2^t - 1$, for all $t < k$. Consider a node with $c_v = k$, and let $u_1, u_2, \ldots, u_k$ denote the vertices eliminated by k in order. On the i-th elimation, the value of c_v was $i - 1$, hence the value of c_{u_i} was at most $i - 1$. Once eliminated, the counter c_{u_i} for node u_i stays unchanged. Hence, by the inductive hypothesis, u_i was responsible for at most $R(i-1)$ other nodes. We then have that

$$R(k) \leq \sum_{t=1}^{k} (R(t-1) + 1) = \sum_{t=0}^{k-1} 2^t = 2^k - 1,$$

establishing the claim. ■

5 Minimal Space Algorithms

In the most restricted case, we have no extra space available. We can refer to such algorithm as *memoryless*, since they cannot store anything about previous events. Can we still obtain reasonable approximations to IS?

We show that there exists a function g allowing us to find a $n/g(d)$-independent set in this model, but that g must now be exponential.

The algorithm RandomDelete given in Fig. 5 selects an endpoint at random from each edge in the stream and removes it from the current solution. Note that RandomDelete is memoryless. For the sake of simplicity, we restrict our attention in this section to the case of graphs.

```
Algorithm RandomDelete
 Input: a stream $E$ of edges

 $V \leftarrow S \leftarrow \emptyset$
 For each edge $e$ in the stream $E$ do
     $V \leftarrow V \cup e$
   Randomly select a vertex $v \in e$
   $S \leftarrow S \cup \{v\}$
 Output $V \setminus S$
```

Figure 5: The algorithm RandomDelete

Intuitively, a memoryless algorithm would seem to be unable to do significantly better than randomly selecting the vertex to be eliminated.

Theorem 5.1 RandomDelete *finds an independent set of expected weight $n/2^{O(d)}$, and this is tight even if the algorithm avoids eliminating vertices unnecessarily.*

Proof: *Upper bound.* Each vertex v belongs to the final solution $V \setminus S$ with probability $2^{-d(v)}$. Therefore, the expected size of the $V \setminus S$ is $\sum_{v \in V} 2^{-d(v)} \geq n/2^d$, using the linearity of expectation and Jensen's inequality.

Lower bound. Consider the graph with vertex set $V = \{v_1, v_2, \cdots, v_n\}$ and edges $\{v_i, v_j\}$ for any $|i - j| \leq k$. Edges arrive in the stream in lexicographic order: (v_1, v_2), (v_1, v_3), $\ldots$, (v_1, v_k), $(v_2, v_3), \ldots, (v_{k+1})$, (v_3, v_4), etc. Note, that all but the first and the last k vertices have degree $2k$. Thus, the average degree $d \leq \Delta = 2k$.

Let I be the independent set constructed by the algorithm. Consider the first vertex v_1. There are two cases, depending on whether v_1 ends up in I.

Case 1: $v_1 \in I$. It means that all the neighbors of v_1 are deleted. The probability of this event is $P[v_1 \in I] = 2^{-k}$. The remaining stream is identical to the original one with $V = V \setminus \{v_1, v_2, \cdots, v_k\}$.

Case 2: $v_1 \notin I$. Suppose v_1 was selected to cover the t-th edge incident on v_1 for some $t \in [1, k]$. Then, the first $t-1$ neighbors of v_1 were selected to cover the first $t-1$ edges incident on v_1 and were deleted as well. The remaining stream is identical to the original one with $V = V \setminus \{v_1, v_2, \cdots, v_t\}$.

Thus, a vertex $v_i \in V$ is inserted in I only in Case 1 and the probability of this event is 2^{-k}, for any $i \in [1, n-k]$. Note, that the last k vertices form a clique, and so only one vertex from this clique contributes to I. Thus, the expected size of the independent set found by the algorithm is at most $\frac{n-k}{2^k} + 1 < \frac{n}{2^k} + 1 < \frac{n}{2^{d/2}}$. ■

Remark. The algorithm has the special property of being *oblivious* in that the solution maintained, or any other part of memory, is never consulted in the operation of the algorithm until it is output.

The utility of advice

When both n and p are known in advance, we can obtain from the BASIC schema an algorithm that requires only logarithmic space in addition to the solution bits. A single bit can record whether a node is contained in the current independent set solution, i.e. in $\overline{S} \cap X$. If $p = 1/d^*$, the reciprocal of the efficient degree, then Lemma 2.1 yields the following bound that is slightly weaker than $i(H)$.

Theorem 5.2 *When n and p are known in advance, there is an online streaming algorithm that finds an independent set of expected weight $\Omega(n/d^*)$ in $O(\log n)$ extra space and using $O(r)$ time to process each edge.*

In comparison with the earlier zero-space algorithms, this suggests that knowledge of the input parameters is highly useful for IS. This relates to the recent *annotation model* of [6], although the assumption there is that the advice is dispensed only after the stream is given.

6 Open questions

What is the right model for graph problems in the streaming context? All of our algorithms for IS use: A read-once-only tape + A tape for the output stream with very limited access + Poly-logarithmic work space. Is poly-logarithmic work space + restricted storage types the way to capture a range of graph problems that do not fit conveniently into existing streaming models? If other graph problems can be also captured by this model, this could grow into a new brand of research.

It would be interesting to compute the constants hidden in our performance gaurantees. Finally, we conjecture that we can make all of our algorithms run without advance knowledge of n.

Acknowledgement We are grateful to Páll Melsted for helpful discussions.

References

[1] K.J. Ahn and S. Guha. Graph Sparsification in the Semi-streaming Model. *ICALP*, 328–338, 2009.

[2] N. Alon, U. Arad and Y. Azar. Independent Sets in Hypergraphs with Applications to Routing via Fixed Paths. *APPROX*, 16–27, 1999.

[3] N. Alon, Y. Matias and M. Szegedy. The space complexity of approximating the frequency moments. *J. Computer and System Sciences* 58(1): 1167–1181, 1999.

[4] N. Alon, J. H. Spencer. *The probabilistic method.* John Wiley and Sons, 1992.

[5] Z. Bar-Yossed, R. Kumar and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. *SODA*, 623–632, 2002.

[6] G. Cormode, M. Mitzenmacher and J. Thaler. Streaming graph computations with a helpful advisor. arXiv:1004.2899v1.

[7] C. Demetrescu, I. Finocchi and A. Ribichini. Trading off space for passes in graph streaming problems. *SODA*, 714–723, 2006.

[8] Y. Emek, M. M. Halldórsson, Y. Mansour, B. Patt-Shamir, and D. Rawitz. Online set packing. To appear in *PODC*, 2010.

[9] L. Epstein, A. Levin, J. Mestre, and D. Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. *STACS*, 2010.

[10] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri and J. Zhang. On Graph Problems in a Semi-Streaming Model. *Theoretical Computer Science*, 348(2): 207–216, 2005.

[11] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri and J. Zhang. Graph distances in the data-stream model. *SIAM J. Comput.* 38(5):1709-1727, 2008.

[12] M. M. Halldórsson, Kazuo Iwama, Shuichi Miyazaki, and Shiro Taketomi. Online independent sets. *Theoretical Computer Science*, 289(2): 953–962, 2002.

[13] S. Muthukrishnan. Data Streams: Algorithms and Applications. Manuscript, *http://athos.rutgers.edu/~muthu/stream-1-1.ps*, 2003.

[14] H. Shachnai and A. Srinivasan. Finding Large Independent Sets of Hypergraphs in Parallel. *SIAM J. Discrete Math.*, 18(3):488–500, 2005.