Trading off space for passes in graph streaming problems

Camil Demetrescu † Irene Finocchi ‡ Andrea Ribichini §

Abstract

Data stream processing has recently received increasing attention as a computational paradigm for dealing with massive data sets. While major progress has been achieved for several fundamental data sketching and statistics problems, there are many problems that seem to be hard in a streaming setting, including most classical graph problems. Relevant examples are graph connectivity and shortest paths, for which linear lower bounds on the “space × passes” product are known. Some recent papers have shown that several graph problems can be solved with one or few passes, if the working memory is large enough to contain all the vertices of the graph. A natural question is whether we can reduce the space usage at the price of increasing the number of passes. Surprisingly, no algorithm with both sublinear space and passes is known for natural graph problems in classical streaming models.

Motivated by technological factors of modern storage systems, some authors have recently started to investigate the computational power of less restrictive streaming models. In a FOCS’04 paper, Aggarwal et al. have shown that the use of intermediate temporary streams, combined with the ability to reorder them at each pass for free, yields enough power to solve in polylogarithmic space and passes a variety of problems, including graph connectivity. They leave however as an open question whether problems such as shortest paths can be solved efficiently in this more powerful model.

In this paper, we show that the “streaming with sorting” model by Aggarwal et al. can yield interesting results even without using sorting at all: by just using intermediate temporary streams, we provide the first effective space-passes tradeoffs for natural graph problems. In particular, for any space restriction of $s$ bits, we show that single-source shortest paths in directed graphs with small non-negative integer edge weights can be solved in $O((n \log^{3/2} n) / \sqrt{s})$ passes. This is the first known streaming algorithm for shortest paths in directed graphs. For undirected connectivity, we devise an $O((n \log n)/s)$ passes algorithm. Both problems require $\Omega(n/s)$ passes under the restrictions we consider. We also show that the model where intermediate temporary streams are allowed can be strictly more powerful than classical streaming for some problems, while maintaining all of its hardness for others.

---

*Work supported in part by the Italian MIUR Project ALGO-NEXT “Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments”.

†Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, Roma, Italy. Email: demetres@dis.uniroma1.it. URL: http://www.dis.uniroma1.it/~demetres.

‡Dipartimento di Informatica, Università di Roma “La Sapienza”, Roma, Italy. Email: finocchi@di.uniroma1.it. URL: http://www.dsi.uniroma1.it/~finocchi.

§Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, Roma, Italy. Email: ribichini@dis.uniroma1.it. URL: http://www.dis.uniroma1.it/~ribichini.
1 Introduction

The typical data size of a wide range of applications in computational sciences can easily reach the order of Terabytes or even Petabytes. In all such applications managing massive data sets, using secondary and tertiary storage devices is a practical and economical way to store and move data: such large and slow external memories, however, are best optimized for sequential access, and thus naturally produce huge streams of data that need to be processed in a small number of sequential passes. Typical examples include data access to database systems [13] and analysis of Internet archives stored on tape [17]. Information naturally occurs in the form of huge data streams also in applications that monitor in real-time network traffic, on-line auctions, transaction logs such as Web usage logs, telephone call records or automated bank machine operations [12, 13, 23]. Among the computational models that have been proposed to deal with massive data sets, data stream processing has therefore received an ever increasing attention in the last few years.

In the classical data stream model [17, 20, 21], input data can be accessed sequentially in the form of a data stream, and need to be processed using a working memory that is small compared to the length of the stream. The main parameters of the model are the number \( p \) of sequential passes over the data and the size \( s \) of the working memory (in bits): throughout this paper, we will refer to the class of problems solvable within \( p \) passes using working memory \( s \) as \( \text{Stream}(p, s) \). A typical additional parameter is the per-item processing time, which should also be kept small. Despite the heavy restrictions of the \( \text{Stream} \) model, major success has been achieved for several data sketching and statistics problems, where \( O(1) \) passes and polylogarithmic working space have been proven to be enough to find approximate solutions (see, e.g., [3, 10, 11] and the extensive bibliographies in [5, 21]). On the other hand, many other problems seem to be far from being solved within similar bounds, including most classical graph problems. Relevant examples are graph connectivity and shortest paths, for which linear lower bounds on \( p \times s \) are known [17]. Some recent papers show that several graph problems can be solved with one or few passes in the semi-streaming model [8, 9, 19] where the working memory size is \( O(n \cdot \text{polylog } n) \) for an input graph with \( n \) vertices: in other words, akin to semi-external memory models [1, 25] there is enough space to store vertices, but not edges of the graph. While \( O(n \cdot \text{polylog } n) \) space seems to be a “sweet spot” for streaming graph problems [21], a natural question already posed in [17, 20] is whether we can reduce the space usage at the price of increasing the number of passes. Surprisingly, to the best of our knowledge no algorithms with both sublinear space and passes are known for natural graph problems in the \( \text{Stream} \) model. Finding effective space-passes tradeoffs in this context appears therefore to be a challenging research direction for both its theoretical and practical implications. Consider for instance the problem of processing a very large graph stored on a file in secondary memory. On a standard computing platform with 1 GB of available main memory, a tradeoff algorithm that runs in \( p = (n \log n)/s \) passes\(^1\) can process a graph with 4 billion vertices and 6 billion edges stored in a 50 GB file in less than 16 passes in the worst case. Using a RAID disk with 100 MB/sec sequential access rate, this would take roughly 2.5 hours. With a streaming algorithm that requires \( s \geq n \log n \) bits without being able to trade space for passes, we would simply not be able to solve the problem at hand (even with infinite time) unless 16 GB of main memory are available.

Motivated by technological factors, some authors have recently started to investigate the computational power of less restrictive streaming models. Today’s computing platforms are equipped with large and inexpensive disks highly optimized for sequential read/write access to data, and among the primitives that can efficiently access data in a non-local fashion, sorting is perhaps the most optimized and well understood. These considerations have led Aggarwal et al. [2] to introduce the “streaming and sorting” model, denoted here as \( \text{StreamSort} \). This model extends \( \text{Stream} \) in two ways: the ability to write intermediate temporary streams and

\(^1 \)Throughout this paper, we assume that all logarithms are to the base of 2.
the ability to reorder them at each pass for free. A StreamSort algorithm alternates streaming and sorting passes: a streaming pass, while reading data from the input stream and processing them in the working memory, produces items that are sequentially appended to an output stream; a sorting pass consists of reordering the input stream according to some (global) partial order and producing the sorted stream as output. Streams are pipelined in such a way that the output stream produced during pass $i$ is used as input stream at pass $(i+1)$. As shown in [2, 22], the combined use of intermediate temporary streams and of a sorting primitive yields enough power to solve efficiently (within polylogarithmic passes and polylogarithmic memory) a variety of problems, including graph connectivity, minimum spanning tree, and geometrical problems. It remains however an open question whether problems such as shortest paths (and even breadth first search) can be solved efficiently in this more powerful model.

**Our contributions.** In this paper we show that the StreamSort model can yield interesting results even without using sorting passes at all: by just using intermediate temporary streams, we provide the first effective space-passes tradeoffs for natural graph problems. Namely, if we denote by W-Stream this more restrictive model without a sorting primitive [22], we show that for any space restriction of $s$ bits:

- Undirected connectivity can be solved in W-Stream by a deterministic algorithm in $O((n \log n)/s)$ passes. By adapting classical communication complexity arguments previously used in the Stream model, we can prove an $\Omega(n/s)$ lower bound on the number of passes for connectivity also in W-Stream. Our algorithm is thus almost optimal.

- Single-source shortest paths in directed graphs with non-negative integer edge weights up to $C$ can be solved in W-Stream by a randomized algorithm in $O((C n \log^{3/2} n)/\sqrt{s})$ passes. This is the first known algorithm for shortest paths on directed graphs in a streaming model. We remark that previous results on distances in streaming models are based on the computation of graph spanners, and these yield approximate distances in undirected graphs only [9]. We also note that the lower bound for connectivity implies an $\Omega(n/s)$ lower bound on the number of passes also for single-source shortest paths.

We remark that for these problems we have exactly the same lower bounds on $p \times s$ in both Stream and in W-Stream. The only known upper bounds in Stream assume $s = \Theta(n \log n)$. On the other hand, our W-Stream algorithms adapt to the available working memory, yielding a full range of possible space-passes tradeoffs. This implies that either W-Stream is more powerful than Stream for these problems, or the known lower bounds in Stream are not tight. This motivates us to conclude the paper with some observations related to the computational power of W-Stream.

A first natural question is whether the use of temporary streams always makes W-Stream more powerful than Stream in a multi-pass setting. In this paper, we give a negative answer, by providing examples of problems that are as hard in W-Stream as in Stream for a given space restriction, regardless of the number of passes. One such example is the classical element distinctness problem, where the challenge is to determine where a given input stream contains any duplicates. This hardness result can be proved using classical tools from communication complexity. We remark that this kind of arguments can be applied to both Stream and W-Stream, but not to StreamSort.

Intuitively, however, the use of intermediate temporary storage should make W-Stream more powerful than Stream, at least for some problems. We exemplify two different ways in which this is indeed the case. We first observe that, from a classical space complexity perspective, intermediate streams can be thought of as part of the algorithm’s working memory. So clearly there can be problems impossible to solve in Stream with a given space restriction, but solvable in W-Stream in a finite number of passes. The recognition of context-free languages is one prominent example. As a second observation, we note that in W-Stream the size of intermediate streams can vary from pass to pass, while in Stream the same input stream is
read at each pass. Counting the total number of processed stream items, rather than the number of passes, may therefore be a more accurate measure for comparing algorithms in the two models. Based on this observation, we show that there are problems for which the number of processed items in W-Stream can be asymptotically smaller than in Stream.

Throughout this paper, we will refer to the class of problems solvable in W-Stream within \( p \) passes using working memory \( s \) as \( W-Stream(p,s) \).

2 Trading off space for passes

In this section we show that using intermediate temporary streams we can achieve the first effective space/passes trade-offs for fundamental graph problems such as undirected connectivity and directed shortest paths. We assume that the input graph is given as an adjacency stream \([6]\), i.e., as a stream \( \Sigma \) of edges in arbitrary order, with \( |\Sigma| = m \).

2.1 Graph connectivity

Let \( G = (V,E) \) be an undirected graph with \( n \) vertices and \( m \) edges. The undirected connectivity problem (UCON) asks whether \( G \) is connected.

**Lower bounds.** By a classical communication complexity argument based on a reduction from the bit-vector disjointness problem (see \([17]\)), UCON requires \( \Omega(n/s) \) passes in Stream when the space restriction is \( s \), i.e., \( UCON \notin Stream(o(n/s),s) \). The same communication complexity argument can be adapted to W-Stream:

**Theorem 1** In W-Stream, UCON requires \( p = \Omega(n/s) \) passes with a space restriction \( s \).

**Upper bounds.** We now describe a deterministic algorithm that solves the more general problem of finding the connected components of \( G \) using \( p = O((n \log n)/s) \) passes and space \( s \) in the W-Stream model.

Given an undirected graph \( G = (V,E) \), let \( C(G) = (V,E') \) be the undirected graph on the same vertex set such that \( (u,v) \in E' \) if and only if \( v \) is the representative vertex of the connected component of \( G \) that contains \( u \). We note that \( C(G) \) represents explicitly the connected components of \( G \) as stars around component representatives. If \( L \) is a list of edges, we denote by \( G_L = (V_L,L) \) the graph induced by edges in \( L \). Thus, \( G_\Sigma = G \).

The algorithm works as follows. Each intermediate stream \( \Sigma_i \) produced by the algorithm is divided into two consecutive parts \( A_i \) and \( B_i \) such that \( G_{B_i} \) is a collection of stars, and \( G_{A_i} \cup G_{B_i} \) has the same connected components as \( G \). At the beginning, \( A_0 = \Sigma \) and \( B_0 = \emptyset \), and thus \( G_{A_0} = G \) and \( G_{B_0} = \emptyset \). At the end, \( G_{A_p} = \emptyset \) and \( G_{B_p} = C(G) \) is the desired result. The generic pass \( i \) of the algorithm works in four phases:

1. Read a prefix \( H \) of edges from \( A_i \) and store in main memory \( M \) each newly encountered vertex until either \( M \) gets full, or all edges of \( A_i \) have been read. Let \( G_H = (V_H,H) \subseteq G_{A_i} \) be the graph induced by the edges in the prefix \( H \) of \( A_i \) read in this phase. As edges are streamed in, also form in \( M \) the connected components of \( G_H \), e.g., by building a spanning forest. No output items are produced during this phase.

2. Read all remaining edges from \( A_i \) (if any). Let \( c(v) \) be the representative vertex of the connected component of \( G_H \) that contains \( v \), if \( v \in V_H \), and let \( c(v) = v \) otherwise. For each input item \( (u,v) \) read from \( A_i \) such that \( c(u) \neq c(v) \), write \( (c(u),c(v)) \) as output item to \( A_{i+1} \).

3. Read all edges from \( B_i \). For each input edge \( (u,v) \) read from \( B_i \), write \( (u,c(v)) \) as output edge to \( B_{i+1} \).
The algorithm repeats the generic pass described above until $A_i$ gets empty. We note that phase 1 is a memory loading phase, which stores $V_H$ in main memory along with a sparse certificate of connectivity of $G_H$ (e.g., a spanning forest). Phase 2 produces an output graph $G_{A_{i+1}}$ obtained from $G_A$ by contracting each connected component of $G_H$ into its representative vertex. Vertices that are in $G_A$, but disappear from $G_{A_{i+1}}$ due to the contraction are put in $G_{B_{i+1}}$ by connecting them to their component representatives in $G_H$. These representatives may be later replaced by newer representatives in successive executions of phase 3 so as to maintain the invariant that $G_{B_{i+1}}$ is a collection of stars. The example of Figure 1 illustrates the effects of one pass of the algorithm.

**Analysis.** To prove the correctness of the algorithm, it suffices to check that the following invariant is maintained at each pass.

**Invariant 1** For each $i \in \{0, \ldots, p\}$, $G_{B_i}$ is a collection of stars, and $G_{A_i} \cup G_{B_i}$ has the same connected components as $G$.

**Proof (sketch).** We prove our claim by induction on the number of passes performed by the algorithm. The base for $i = 0$ is straightforward, since $G_{A_0} = G$ and $G_{B_0} = \emptyset$. We assume by inductive hypothesis that the invariant holds at pass $i$, and we show that it also holds at pass $i + 1$. First, observe that $G_{B_{i+1}}$ is obtained in phases 3 and 4 as union of stars from $G_{B_i}$ and stars that represent the connected components of $G_H$. If the stars produced in the two phases are not disjoint, the union may not be a collection of stars. For this reason, if a star in $G_{B_i}$ intersects a component of $G_H$, its center is replaced in phase 3 by its representative in $G_H$. Thus, $G_{B_{i+1}}$ is a collection of stars. To prove that $G_{A_{i+1}} \cup G_{B_{i+1}}$ has the same connected components as $G$, observe that each connected component of $G_H \subseteq G_{A_i} \subseteq G_{A_{i+1}} \cup G_{B_i}$ is replaced by a star in $G_{A_{i+1}} \cup G_{B_{i+1}}$, and thus connectivity information is maintained. □

Assuming that the main memory $M$ has a size of $s$ bits, we now show that the algorithm terminates in at most $p = O((n \log n)/s)$ passes.

**Theorem 2** In W-Stream, $UCON$ can be solved with $p = O((n \log n)/s)$ passes when the space restriction is $s$.

**Proof (sketch).** For the sake of simplicity, we assume that the input graph $G$ contains no self-loops. Notice that during pass $i$, all vertices of $V_H$ that are not representatives of connected components of $G_H$ disappear from $G_{A_{i+1}}$ (phase 2). Since $G_H$ is induced by a set of edges, each connected component of $G_H$ contains at least two vertices. Thus, there are at least $V_H/2$ vertices in $G_{A_i}$ that disappear from $G_{A_{i+1}}$, so in at most $p \leq 2n/V_H$ passes $G_{A_p}$ gets empty. Since phase 1 fills memory $M$ with vertices and a spanning forest of $G_H$ until it gets full, and storing each vertex label requires $\log n$ bits of space, then $|V_H| = \Theta(s/\log n)$. This implies that $p = O((n \log n)/s)$. □
2.2 Single-source shortest paths

Let $G = (V, E, w)$ be an edge-weighted directed graph with $n$ vertices and $m$ edges. In the single-source shortest paths problem (SSSP), we wish to find distances from a given source $t$ to all the other vertices in $G$. In this section, we assume that each edge $(u, v) \in E$ is represented in the input stream $\Sigma$ as a triple $(u, v, w_{uv})$, where $w_{uv}$ is the weight of the edge. In the following, we assume that each vertex label and each edge weight can be represented with $\log n$ bits.

We say that the weight of a path is the sum of the weights of its edges. The distance $\text{dist}_{xy}$ between two vertices $x$ and $y$ of the graph is the weight of a minimum weight path connecting them.

**Lower bounds.** We first observe that, since UCON can be reduced to SSSP, then the lower bound for UCON given in Theorem 1 also holds for SSSP:

**Theorem 3** In W-Stream (and thus in Stream), SSSP requires $p = \Omega(n/s)$ passes when the space restriction is $s$.

This implies that, if we want to achieve sublinear space $s = o(n)$, then $p = \omega(1)$ passes are required. And yet, if we stick to constant number of passes, the situation can be even worse. Feigenbaum et al. have proved in [9] a higher lower bound in the Stream model: the lower bound implies that finding vertices up to distance $d = O(1)$ from a given source in less than $d$ passes requires $\Omega(n^{1+1/2d})$ space. Since $p$ is constant and W-Stream can be simulated in Stream at the price of increasing the size of the working memory by a factor of $p$ (see [22]), it is not difficult to see that this result also holds in W-Stream. This confirms that space efficient algorithms for SSSP in both Stream and W-Stream always require multiple passes.

We also remark that finding efficient streaming algorithms for the simpler problem of breadth-first traversal of a graph has been posed as an open problem even in the StreamSort model [2].

**Upper bounds.** In the remainder of this section, we devise the first algorithm for single-source shortest paths in directed graphs in a streaming model. In particular, we prove the following theorem:

**Theorem 4** In W-Stream, directed SSSP with non-negative integer edge weights up to $C$ can be solved with $p = O((C \cdot n \cdot \log^{3/2} n)/\sqrt{s})$ passes when the space restriction is $s$. Distances produced by the algorithm are correct with probability at least $1 - 1/n^\beta$ for any positive constant $\beta$. The size of each intermediate stream is $O(m + n \cdot \sqrt{s}/\log n)$.

Notice that, for $C = o(\sqrt{s}/\log^{3/2} n)$ we can get both $p$ and $s$ sublinear in $n$.

**Overview of the algorithm.** A typical issue in streaming settings where edges are given in arbitrary order is that following a path seems to require in the worst case as many passes as its length. Finding long paths may therefore require lots of passes. To overcome this difficulty, we argue that, if we were able to find long paths as the concatenation of short paths built “in parallel” within the same passes, this would result in a substantial reduction of the worst-case number of passes required to follow a path of arbitrary length. Similarly to previous algorithms for path problems in parallel and dynamic settings (see, e.g., [16, 24]), the main idea of our algorithm is to perform many short searches from a random subset of vertices of the graph “in parallel”. This yields short distances in the graph. To find longer distances, the algorithm stitches together short paths. Using a probabilistic argument, we can prove that distances obtained in this way are correct with high probability.

**Finding distances up to $\ell$.** Let $A = \{c_1, c_2, \ldots, c_A\}$ be a subset of vertices of the graph and let $\ell > 0$ be an integer parameter. As a first ingredient for solving SSSP in W-Stream, we describe a procedure $\text{shortDist}(A, \ell)$ that finds the distances from each source $c_j \in A$ to all
other vertices that are up to distance \( \ell \) from \( c_j \) in \( p = O(\frac{n |A| \log n}{s} + \ell) \) passes in a graph with non-negative integer edge weights.

Our procedure is a multi-source, bounded depth, streamed implementation of Dijkstra’s algorithm [7], where the “priority queue” is maintained implicitly on intermediate streams. Let \( \{\gamma_1, \gamma_2, \ldots, \gamma_q\} \) be a partition of the input stream \( \Sigma_0 = \Sigma \) into the minimum number \( q \) of groups \( \gamma_i \) such that: (1) all edges in a group \( \gamma_i \) share the same end vertex \( y_i \), i.e., \( \gamma_i = (a, y_i, w_{a y_i}) (b, y_i, w_{b y_i}) \cdots (z, y_i, w_{z y_i}) \); and (2) the concatenation of groups \( \gamma_i \) yields \( \Sigma_0 \), i.e., \( \Sigma_0 = \gamma_1 \gamma_2 \cdots \gamma_q \). Notice that, if the edges in the input stream \( \Sigma \) were ordered by their end vertex, there would exist one unique group \( \gamma_i \) per vertex. In general, the same end vertex may be shared by more than one group, i.e., it may be \( y_a = y_b \) with \( a \neq b \).

Each intermediate stream \( \Sigma_h \), with \( h > 0 \), created by the algorithm has the form \( \Sigma_h = \gamma_1 \delta_1 \gamma_2 \delta_2 \cdots \gamma_q \delta_q \), where \( \delta_i = (d_{i1}, f_{i1})(d_{i2}, f_{i2}) \cdots (d_{i|A|}, f_{i|A|}) \). For each pair \((d_{ij}, f_{ij}) \in \delta_i\), \( d_{ij} \) is an upper bound to the distance \( \text{dist}_{c_i y_i} \) from \( c_j \in A \) to \( y_i \) and flag \( f_{ij} \) is true if and only if \( y_i \) is settled w.r.t. \( c_j \), i.e., \( \text{dist}_{c_j y_i} \) has been correctly determined by the algorithm. In the first pass, the algorithm lets \( f_{ij} = \text{false} \) for each \( i \) and \( j \); it also lets \( d_{ij} = 0 \) if \( c_j = y_i \), and \( d_{ij} = +\infty \) otherwise. The goal of successive passes is to progressively decrease each \( d_{ij} \) to the weight of a minimum weight path from \( c_j \) to \( y_i \) that goes through one of the edges in \( \gamma_i \).

The core loop of algorithm shortDist alternates extraction and relaxation passes. During an extraction pass, the algorithm loads in main memory, for each \( c_j \in A \), a pool \( P_{c_j} \) of at most \( k = s/(|A| \cdot \log n) \) vertices \( v \) together with their exact distance \( \text{dist}_{c_j v} \) from \( c_j \) (after the first extraction pass, each pool \( P_{c_j} \) includes only vertex \( c_j \) and \( \text{dist}_{c_j c_j} = 0 \)). During a relaxation pass, the algorithm improves the distance upper bounds \( d_{ij} \) using edges in \( \gamma_i \) that emanate from \( P_{c_j} \). In more details:

- **Extraction pass.** Let \( d_{ij}(v) = \min_{v = y_k}(d_{ij}) \) be the priority of \( v \) w.r.t. \( c_j \). For each \( c_j \in A \), load in \( P_{c_j} \) up to \( k \) unsettled vertices with the same minimum priority w.r.t. \( c_j \), if it does not exceed \( \ell \). For each vertex \( v \) in \( P_{c_j} \), it holds \( \text{dist}_{c_j v} = d_{ij}(v) \). When all \( P_{c_j} \)’s get empty, the algorithm halts.

- **Relaxation pass.** For each \( i = 1, 2, \ldots, q \), decrease each \( d_{ij} \) in the output \( \delta_i \) to the weight of a minimum weight path from \( c_j \) to \( y_i \) that goes through one of the edges in \( \gamma_i \) that emanate from \( P_{c_j} \). Also, make vertices in each \( P_{c_j} \) settled w.r.t. \( c_j \) by letting \( f_{ij} \leftarrow \text{true} \) in the output stream for each \( y_i \in P_{c_j} \).

We remark that all the vertices that are in each pool \( P_{c_j} \) at the end of an extraction pass have exactly the same distance from \( c_j \). All these vertices would be extracted from the priority queue in consecutive iterations of a classical implementation of Dijkstra’s algorithm with source \( c_j \). When the algorithm is over, for each \( c_j \) and each vertex \( v \) that is settled w.r.t. \( c_j \), the distance \( \text{dist}_{c_j v} \) is implicitly encoded in the output stream as \( \min_{v = y_k}(d_{ij}) \) and can be easily made explicit with a simple post-processing in \( O((n \log n)/s) \) passes.

**Analysis.** We now discuss the number of passes required by algorithm shortDist.

**Theorem 5** Algorithm shortDist \((A, \ell) \) runs in \( p = O(\frac{n |A| \log n}{s} + \ell) \) passes using \( s \) bits of working memory and intermediate streams of size \( O(m \cdot |A|) \).

**Proof (sketch).** The algorithm keeps in the working memory up to \( k = s/(|A| \cdot \log n) \) vertices in each of the \( |A| \) pools \( P_{c_j} \). Since storing each vertex label requires \( \log n \) bits, the algorithm uses at most \( k \cdot |A| \cdot \log n = s \) bits of main memory. The bound on the size of intermediate streams follows from the fact that each of them contains \( m + q \cdot |A| \) items and \( q \) can be as high as \( m \) in the worst case.

To bound the number of passes, consider the vertex \( c_j \in A \) such that \( P_{c_j} \) is the last pool to get empty. Let \( P_1, P_2, \ldots, P_k \) be the content of pool \( P_{c_j} \) after successive extraction passes. We say that \( P_i \) is full if it contains exactly \( k \) vertices, and it is incomplete otherwise. Notice that,
since each vertex appears in at most one $P_i$, there can be at most $(n/k)$ full $P_i$’s. We now bound the number of incomplete $P_i$’s. Note that in each set $P_i$, all vertices have the same distance from $c_j$. Denote this distance by $d(P_i)$. Set $P_i$ can be incomplete only if $d(P_i) < d(P_{i+1})$, or $i = t$. Since $d(P_i) \leq \ell$ and edge weights are non-negative integers, there can be at most $\ell$ passes $i$ such that $d(P_i) < d(P_{i+1})$, and thus at most $\ell$ $P_i$’s can be incomplete. Hence, the total number of passes $i$ of $P_i$’s cannot exceed $(n/k + \ell)$. Since the algorithm generates a new $P_i$ every two passes in the core loop, then it performs a total number of $p = O(t) = O(n/k + \ell) = O(n[A\log n]/s + \ell)$ passes.

Notice that for $A = \{t\}$ algorithm $\text{shortDist}$ solves $\text{SSSP}$ up to distance $\ell = O((n \log n)/s)$ from a given source $t$ in $O(n \log n)/s$ passes. By Theorem 3, this bound is optimal in $W$-Stream up to a log factor.

**Reducing the size of intermediate streams.** We now show how to reduce the size of intermediate streams to $O(m + n \cdot |A|)$. The main idea is to preprocess the input stream so as to reduce the number $q$ of groups $\gamma_i$ before starting the $\text{shortDist}$ algorithm. To this aim, we simply partition the input stream $\Sigma$ into $\max\{1, m/(n \cdot |A|)\}$ subsequences of size $\leq n \cdot |A|$ each, and we reorder edges $(x,y,w_{xy})$ in each subsequence by end vertex $y$. This can be done in $O(n \cdot |A| \log n)/s$ passes by using a $W$-Stream variant of the sorting algorithm described in [20]. The preprocessing can thus be performed within the same asymptotic number of passes as $\text{shortDist}$. Notice that the number of groups $\gamma_i$ in each reordered subsequence cannot be larger than $n$. Hence, the total number $q$ of groups $\gamma_i$ in the whole preprocessed stream given as input to $\text{shortDist}$ will not exceed $n \cdot \max\{1, m/(n \cdot |A|)\} = \max\{n, m/|A|\}$. The size of each intermediate stream in $\text{shortDist}$ will therefore be $m + q \cdot |A| \leq m + \max\{n \cdot |A|, m\} = O(m + n \cdot |A|)$ as claimed.

**Finding all distances from a given source.** We now describe an algorithm $\text{sssp}(G,t)$ that solves $\text{SSSP}$ with source $t$ in a graph $G = (V,E,w)$ with $n$ vertices and non-negative integer edge weights up to $C$ in $O((nC\log^{3/2} n)/\sqrt{s})$ passes, assuming a space restriction of $s$ bits in the $W$-Stream model. Algorithm $\text{sssp}(G,t)$ works as follows:

1. Pick a subset $A \subseteq V$ of $\sqrt{s/\log n}$ vertices, including source $t$. All vertices but $t$ are chosen uniformly at random.
2. Find distances up to $\ell = (\alpha Cn \log^{3/2} n)/\sqrt{s}$ in $G$ from each of the vertices in $A$, where $\alpha$ is any constant $> 1$.
3. Build a weighted compressed graph $G^* = (A,E^*,w^*)$ on vertex set $A$ such that there is an edge $(c_1,c_2) \in E^*$ with weight $w_{c_1c_2}^* = \text{dist}_{c_1c_2}$ if and only if $\text{dist}_{c_1c_2} \leq \ell$.
4. Compute distances $\text{dist}_{ct}^*$ from $t \in A$ to all other vertices $c \in A$ in $G^*$.
5. For each vertex $v \in V$ whose distance from $t$ has not been determined in step 2 being higher than $\ell$, compute it as $\text{dist}_{tv} = \min_{c \in A}\{\text{dist}_{tc}^* + \text{dist}_{cv}\}$.

Before describing a $W$-Stream implementation of $\text{sssp}$, we prove that each distance larger than $\ell$ computed by the algorithm is correct with high probability, assuming that distances up to $\ell$ computed in step 2 are correct.

**Theorem 6** Each distance $\text{dist}_{tv} > \ell$ computed by algorithm $\text{sssp}$ is correct with probability at least $1 - 1/n^{\alpha - 1}$.

**Proof (sketch).** If $\text{dist}_{tv} > \ell$, then it is obtained in step 5 as $\text{dist}_{tv} = \min_{c \in A}\{\text{dist}_{tc}^* + \text{dist}_{cv}\}$. Since edge weights of $G$ are $\leq C$, then any shortest path from $t$ to $v$ in $G$ will necessarily contain at least $r = \ell/C$ edges. Let $\pi_{tv}$ be any shortest path from $t$ to $v$. Adapting to our setting a well known sampling theorem from [14], we now show that, with high probability, every subpath of $\pi_{tv}$ with $r$ vertices contains at least a vertex from $A$. Consider one of those subpaths, and
let $Q$ the probability that it does not contain vertices of $A$. Since a vertex of $G$ belongs to $A$ with probability $|A|/n$, then:

$$Q = \left(1 - \frac{|A|}{n}\right)^r < 2^{-\frac{|A|}{n^a}} = \frac{1}{n^a}.$$ 

Since there can be at most $n/r \leq n$ disjoint subpaths of $\pi_{tv}$ with $r$ vertices, then the probability that each of them contains a vertex of $A$ is at least $1 - Q \cdot n > 1 - 1/n^{a-1}$. This implies that, with probability at least $1 - 1/n^{a-1}$, $\pi_{tv}$ can be broken into the concatenation of subpaths of at most $r$ vertices of the form $\pi_{c_i,c_j}$, with $c_i, c_j \in A$, plus one final subpath of the form $\pi_{c^*,v}$, where $c^*$ minimizes $\min_{c \in A} \{ \text{dist}_{tc}^* + \text{dist}_{cv} \}$. Since w.h.p. each $\pi_{c_i,c_j}$ is a shortest path with weight at most $\ell$, then it corresponds to an edge $(c_i, c_j) \in E^*$. Thus, w.h.p. the value $\text{dist}_{tc^*}$ computed in step 4 is the correct distance from $t$ to $c^*$. To conclude the proof, observe that $\text{dist}_{c^*v}$ has also weight at most $\ell$ w.h.p., and thus it has been correctly determined in step 2. Thus, $\text{dist}_{tv} = \text{dist}_{tc^*} + \text{dist}_{c^*v}$ with probability at least $1 - 1/n^{a-1}$. 

**Implementation.** We now sketch how steps 1–5 of algorithm sssp can be implemented in W-Stream:

1. As $|A| = \sqrt{s/\log n}$, then vertices of $A$ can be sampled and maintained in main memory.
2. To find distances up to $\ell$ from each vertex in $A$, we can just run algorithm shortDist$(A, \ell)$ described earlier in this section. The algorithm stores distances on the output stream.
3. Graph $G^*$ can be stored in main memory, since it requires no more than $|A|^2 \log n = s$ bits. To build it, we can just make one pass and build an $|A| \times |A|$ weight matrix $w^*$ such that for each $c_j, c \in A$ $w^*_j,c = \min_{c \in G} \{d_{ij}\}$.
4. Distances $\text{dist}_{tc}$ can be computed by running any internal-memory single-source shortest paths algorithm on $G^*$ with source $t$, and can be stored in main memory.
5. Compute final distances for $s/\log n$ vertices $K \subseteq V$ at a time. For each $K$, compute in one pass $\text{dist}_{tkv} = \min_{i,j : v \in K} \{ \text{dist}_{tc_j}^* + d_{ij} \}$ for each $v \in K$. At the end of the pass, flush computed distances to the output stream in any desired format.

**Analysis.** We now discuss the time and space requirements of algorithm sssp.

**Theorem 7** The number of passes required by algorithm sssp is $O((nC \log^{3/2} n)/\sqrt{s})$, using $s$ bits of main memory and intermediate streams of size $O(m + n \cdot \sqrt{s/\log n})$ in the W-Stream model.

**Proof (sketch).** Steps 1 and 4 are entirely performed in main memory, and thus require no streaming passes. Step 3 requires one pass and step 5 takes $O((n \log n)/s)$ passes. The entire procedure is dominated by the number of passes of algorithm shortDist in step 2, which is $O\left(n |A| \log n + \ell \right) = O\left(nC \log^{3/2} n/\sqrt{s}\right)$ by Theorem 5. The size of intermediate streams also follows from Theorem 5 and from the preprocessing technique described thereafter.

## 3 Separation and hardness results

The algorithms presented in Section 2 show that the ability to write intermediate streams makes it possible to obtain, at least for some problems, a full range of possible space/passes tradeoffs, whereas the only known upper bounds in Stream assume $s = \Theta(n \log n)$. These results naturally raise the question of whether W-Stream is more powerful than Stream. In this section we therefore study some aspects related to the computational power of W-Stream. We first exemplify problems in W-Stream that are impossible to solve in Stream for a given
space restriction and problems that require a smaller number of processed items in $W$-Stream than in $Stream$. Note that in $W$-Stream the size of intermediate streams can vary from pass to pass, while in $Stream$ the same input stream is read at each pass: counting the total number of processed stream items, rather than the number of passes, may therefore be a more accurate measure for comparing algorithms in the two models. On the other hand, we also provide examples of problems that are as hard in $W$-Stream as in $Stream$ for a given space restriction (regardless of the number of passes). We obtain this result by adapting to $W$-Stream classical communication-complexity arguments used for proving lower bounds in $Stream$. This kind of arguments cannot instead be applied to $StreamSort$.

**Breaking the space wall.** From a space complexity perspective, intermediate streams can be thought of as part of the algorithm’s working memory. It is therefore conceivable that one should be able to solve problems in $W$-Stream with a space restriction that would make them unsolvable in $Stream$. Consider, for instance, the following Parenthesis Language Recognition problem (PLR): let $L$ be the context-free parenthesis language $\{S \rightarrow (\mid ) \mid (S) \mid SS\}$, let $x$ be a string of $n$ symbols in $\{(,\})$ represented as a data stream, find out if $x \in L$. We first prove that PLR cannot be solved in $Stream$ using less than logarithmic working memory:

**Lemma 1** $PLR \notin Stream(p, o(\log n))$, for any number $p$ of passes.

**Proof (sketch).** Since the parenthesis language is a nonregular context-free language, we can use the following result of Alt et al. [4]: if $L$ is a nonregular deterministic context–free language and $L \in NSPACE(s(n))$, then the recognition of $L$ requires space $s(n) \geq c \cdot \log n$ for some constant $c$ and infinitely many $n$. Clearly, this lower bound also applies to $Stream$ algorithms and implies that, independently of the number of passes, PLR cannot be solved using less than logarithmic space.

On the other hand, we can easily solve PLR in $W$-Stream with a constant size working memory, by removing pairs of consecutive matching parentheses from the stream at each pass, until possible, and returning true if the stream gets empty. Hence, $PLR \in W$-Stream$(n, O(1))$. Our first separation result immediately follows from this observation and from Lemma 1:

**Theorem 8** $Stream(n, O(1)) \subset W$-Stream$(n, O(1))$.

**Reducing the number of processed items.** The ability to manipulate the data stream makes it possible, at least for some problems, to discard at each pass items that are no longer useful, thus reducing the overall number of items that an algorithm has to process. Consider, as an example, the following FORK problem: given two vectors $A$ and $B$ of $n$ numbers with $A[1] = B[1]$ and $A[n] \neq B[n]$, find a “fork” index $i$ such that $A[i] = B[i]$ and $A[i+1] \neq B[i+1]$. Assume that $A$ and $B$ are given as an input stream of the form $A[1], A[2], \ldots, A[n], B[1], B[2], \ldots, B[n]$ with items in $\{1, \ldots, n\}$. The following lower bound on the space $\times$ passes product follows from a communication complexity lower bound on FORK by Grigni and Szepesvári [15]:

**Lemma 2** FORK in $W$-Stream (and thus in $Stream$) requires $p \times s = \Omega(\log^2 n)$.

Lemma 2 implies that, if we stick to logarithmic space, then the number of passes of any streaming algorithm solving FORK must be $p = \Omega(\log n)$. Since in $Stream$ we have to process all the items in the input stream at every pass, it follows that the number of processed items of any $Stream$ algorithm must be $\Omega(n \log n)$ when $s = O(\log n)$.

Instead, we can solve FORK in $W$-Stream more efficiently as follows. Consider a simple binary search-like algorithm, recurring upon the following conditions: (1) if $A[n/2] = B[n/2]$, then there must be a fork index in the second half of the vectors; (2) if $A[n/2] \neq B[n/2]$, then there must be a fork index in the first half of the vectors. At each pass, we can thus halve the size of the intermediate stream, just by not copying the uninteresting half of the input stream. It is easy to see that this algorithm uses $O(\log n)$ space, runs in $O(\log n)$ passes, and processes only $O(n)$ items overall. From the considerations above, we get our separation with respect to the number of processed items:

\[9\]
**Theorem 9** FORK can be (optimally) solved in W-Stream with space \( s = O(\log n) \) and \( O(n) \) processed items. This is impossible to achieve in Stream.

**Hardness results.** Even if the use of intermediate streams makes W-Stream more powerful than Stream for some problems, in this section we show that for other problems W-Stream maintains all of the hardness of classical streaming. In particular, we exemplify problems for which the use of intermediate streams does not help at all, except for possibly simplifying the task of designing streaming algorithms.

Ruhl and Aggarwal et al. [2, 22] already observed that, for a small number of passes, intermediate streams do not help much, regardless of the problem considered. Indeed, W-Stream can be simulated in Stream at the price of increasing the size of the working memory by a factor of \( p \): the simulation given in [2, 22] proves that W-Stream\((p, s) \subseteq \text{Stream}(p, p \cdot s)\), making intermediate streams useless when \( p \) is small.

In the following we show that there are problems for which using intermediate streams does not help at all, even regardless of the number of passes. As an example, we take the element-distinctness problem (ED), that requires to find if there are any duplicates in a stream of \( n \) numbers in \( \{1, \ldots, n\} \). We first give a lower bound on the passes \( \times \) space product by adapting to W-Stream the standard communication-complexity based lower bound used in the Stream model [17]. Although very similar to the proof given in [17], we report a complete proof here for the sake of completeness.

**Theorem 10** Any W-Stream algorithm for element distinctness requires \( p \times s = \Omega(n) \).

**Proof (sketch).** Consider the bit-vector-disjointness problem, in which Alice and Bob have two \( n \)-bit-vectors \( A \) and \( B \), respectively, and want to know whether \( A \cdot B > 0 \). This problem can be reduced to ED in the following way. Alice creates a stream containing the indices corresponding to the 1’s in vector \( A \), and Bob does the same for vector \( B \). Then Alice runs a W-Stream algorithm for element-distinctness on her stream, producing an intermediate stream, and when the input stream is over she sends the content of her working memory to Bob. Bob continues to run the same W-Stream algorithm starting from the memory image received from Alice, reading from his own input stream and producing his own intermediate stream. When the stream is over, Bob sends his memory image back to Alice, who starts a second pass by taking as input the intermediate stream that she produced at the previous pass. At the end, the streaming algorithm will determine whether all the elements in the two input streams are distinct or not: notice that the elements are distinct if and only if \( A \cdot B > 0 \), which is exactly the solution to bit-vector-disjointness. If, by contradiction, the total working memory used by Alice and Bob has size \( o(n/p) \), then the total number of bits sent between Alice and Bob in \( p \) passes would be \( o(n/p) \cdot p = o(n) \), which would violate the \( \Omega(n) \) communication complexity lower bound for bit-vector-disjointness [18].

Since there is a folklore Stream algorithm that solves ED with \( p = O(n/s) \) passes, the use of intermediate streams is of no help for this problem.

We remark that arguments similar to the proof of Theorem 10 can be used for proving also other lower bounds in W-Stream. In particular, many classical communication-complexity based lower bounds known in Stream can be adapted to W-Stream, as well. This technique yields, for instance, W-Stream lower bounds for graph problems such as undirected connectivity and shortest paths (see Theorem 1 and Theorem 3 in Section 2).
References


